2007-06

An OSKit-base implementation of least privilege separation kernel memory partitioning

Carter, Donald W.

Monterey, California. Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

AN OSKIT-BASED IMPLEMENTATION OF LEAST PRIVILEGE SEPARATION KERNEL MEMORY PARTITIONING

by

Donald W. Carter

June 2007

Thesis Advisor:	Cynthia E. Irvine
Co-Advisor:	Tim Vidas

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** June 2007	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE An OSKit-Based Implementation of Least Privilege Separation Kernel Memory Partitioning		**5. FUNDING NUMBERS**	
6. AUTHOR(S) Donald W. Carter			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000		**8. PERFORMING ORGANIZATION REPORT NUMBER**	
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A		**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**	
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited		**12b. DISTRIBUTION CODE**	

13. ABSTRACT (maximum 200 words)

In an environment with valuable information assets, the threat of subversion is real. Thus, systems must be built from the ground up to counter the level of sophistication and capital that is pitted against them. To build such systems, rigorous assurance criteria must be met.

Currently for high assurance systems there is no publicly available example of their design and construction. The Trusted Computing Exemplar (TCX) Project is intended to make publicly available a high assurance component and its evaluation evidence. This work is to build a working prototype of selected TCX kernel functionality.

The prototype is constructed and based on OSKit, and restricts information flow between memory partitions and resource accesses made by processes. Pages are statically allocated on a per-partition basis and page faults are handled by the kernel.

The prototype demonstrates a least privilege-based approach to exported resource management. It uses a separation kernel with preloaded configuration data to allocate memory resources to processes.

14. SUBJECT TERMS separation kernel, least privilege, high assurance, paging, OSKit			**15. NUMBER OF PAGES** 101
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

AN OSKIT-BASED IMPLEMENTATION OF LEAST PRIVILEGE SEPARATION KERNEL MEMORY PARTITIONING

Donald W. Carter
Civilian, Naval Postgraduate School
B.S., Cal-State University of Bakersfield, 2005

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 2007

Author: Donald W. Carter

Approved by: Cynthia E. Irvine, Ph.D.
Thesis Advisor

Tim Vidas
Co-Advisor

Peter J. Denning, Ph.D.
Chairman, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

In an environment with valuable information assets, the threat of subversion is real. Thus, systems must be built from the ground up to counter the level of sophistication and capital that is pitted against them. To build such systems, rigorous assurance criteria must be met.

Currently for high assurance systems there is no publicly available example of their design and construction. The Trusted Computing Exemplar (TCX) Project is intended to make publicly available a high assurance component and its evaluation evidence. This work is to build a working prototype of selected TCX kernel functionality.

The prototype is constructed and based on OSKit, and restricts information flow between memory partitions and resource accesses made by processes. Pages are statically allocated on a per-partition basis and page faults are handled by the kernel.

The prototype demonstrates a least privilege-based approach to exported resource management. It uses a separation kernel with preloaded configuration data to allocate memory resources to processes.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I wish to thank my wife and two kids for their loving support during the writing of this thesis. I cannot thank my thesis advisors, Dr. Cynthia Irvine and Tim Vidas, enough for whose tireless efforts and technical assistance throughout the research work of this thesis helped me complete it on time.

This material is based upon work supported by the National Science Foundation under Grant No. DUE - 0414102. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

This material is also based upon work supported by the Office of Naval Research. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the Office of Naval Research.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. MOTIVATION

Finding the right balance between protecting data from those who should not have access and enabling data sharing with those who have a need to know is a difficult problem. This problem is a real concern for those responsible over protecting classified data. To help those responsible, the Principle of Least Privilege (PoLP) [1] can motivate approach to system design that results in the protection of data while still permitting access to those with the access need. This approach is beneficial for building high assurance systems that protect sensitive data.

One problem with high assurance systems is that there is no publicly available example of their design and construction. This does not enable reuse of key concepts associated with the design and construction of high assurance systems, which would benefit future projects in the domain space for both research and commercial use.

The motivation for this thesis is to provide a prototype of certain separation kernel functionality for the TCX Project, which has the goal of providing a high assurance kernel that adheres to the PoLP and makes publicly available its evaluation evidence.

B. PURPOSE

The purpose of the work done and explained in this paper is to prototype certain functions for a Least Privilege Separation Kernel (LPSK) as part of the TCX Project. This thesis focuses on memory separation that adheres to the PoLP [2]-[4].

C. THESIS ORGANIZATION

The organization of this work is as follows:

- Chapter I discusses the motivation and purpose of this thesis.
- Chapter II discusses the background information for understanding the thesis work.

- Chapter III discusses the operating system development kits that were surveyed and chosen for use in the LPSK prototype.
- Chapter IV discusses the implementation of the LPSK prototype.
- Chapter V discusses the procedures used in testing the LPSK prototype and their results.
- Chapter VI concludes with a discussion of the benefits gained from the approach taken and future work still needed to be done.

II. BACKGROUND

This chapter provides background on a range of topics associated with the prototype implementation of a least privilege separation kernel interface. To fully understand this work one must be familiar with the topics of subversion, the TCX project, open source configurable operating systems, separation kernels, and the Least Privilege Separation Kernel (LPSK).

A. SUBVERSION

When other forms of attack against a system are not possible, sophisticated, well funded attackers employ subversion. Even though subversion is time consuming, these attackers are sufficiently focused to mount such long range attacks. To better understand the aspects of subversion, this section will define subversion, describe subversion's purpose and who uses it, explain how subversion is implemented, deployed and triggered, outline what we can do to prevent subversion with its affiliated mitigation costs, and reiterate why subversion prevention is important.

1. Definition of Subversion

System Subversion as defined by Myers is '… the covert and methodical undermining of internal and external controls over a system lifetime to allow unauthorized or undetected access to system resources and/or information.' Covertly compromising a system through subversion permits the use of surreptitious methods to allow undetectable unauthorized access to a system's data or controls [5].

2. Purpose of Subversion

The general purpose of subversion is to covertly gain privileged access to the target system in order to exploit or threaten to exploit it at the most opportune time. Subversion is the attack of choice by the professional attacker [6].

3. Implementation of Subversion

Implementing subversion is tricky, but can be done by one or more compromised insiders and through the use of special mechanisms. Compromising insiders or getting compromised individuals and/or mechanisms inside a government and/or civilian facility is a method that sophisticated adversaries will exploit when necessary. To deploy subversion, a mechanism or insider must get a back door inside a system that is hidden from detection until used. This may be done through the use of malware or rootkitting a system during any of the stages of the life cycle of the system. A trigger for a backdoor might be as simple as receipt of a single UDP packet over the Internet by a system machine that has been subverted [7]. The ability of an insider to deploy such subversion mechanisms makes the insider a great potential threat.

4. Subversion Vulnerability Pertinence

Experiments in system subversion have demonstrated that any medium level programmer has the capability to subvert a system, and it did not take very many lines of code to subvert a system [7]-[10].

5. Protection against Subversion

To mitigate the threat of system subversion, certain guarantees in a system's security functional requirements and security assurance requirements must be established. An international standard known as the Common Criteria has been established as a means to guarantee that functional and assurance requirements are met to the level as promised. The Common Criteria has an assurance ranking system ranging from EAL1 to EAL7 [11]. EAL7 is similar to Class A1 from the Trusted Computer System Evaluation Criteria [12], an older standard. To obtain a high EAL ranking, the system must go through costly and time consuming methodical processes, but considering the harm unauthorized access to information could cause, these extra measures are worth it. The Trusted Computing Exemplar (TCX) project is developing an EAL 7 system that will be openly available [2].

6. Cost of Mitigation

The use of a highly rigorous development methodology, as required at EAL7, is a way to protect a system from subversion. Protecting DoD facilities this way from subversion is quite expensive. System subversion can be mitigated by methodically verifying every aspect of the system through formal methods and other procedures. Unfortunately, this method only works on small systems since "[t]he size and complexity of today's typical large system prohibit attempts to demonstrate that the entire system is verifiable" [6].

7. Subversion Summary

Subversion was defined via Myers' system subversion definition. Subversion's purpose and the people involved are discussed to better understand the threat. Subversion's implementation, deployment and triggering mechanisms were briefly discussed. The lack of readily available development frameworks to build high assurance systems has kept development costs high. To combat these costly endeavors, NPS is developing the TCX project, one objective of which is to reduce the time necessary to generate high assurance systems.

B. TCX PROJECT

1. TCX Approach

The objective of the Trusted Computing Base (TCX) project is to make readily available source code and documentation that will help others build more complex high assurance systems [2]. The project is building an open source high assurance development framework, least-privilege separation kernel, and model application. The intent for the final kernel is to bring the system to an initial secure state and to ensure that every possible subsequent state is secure.

2. Implementation of TCX Approach

The implementation of the TCX project involves many stages. The first stage consists of defining a high assurance development framework. The second stage involves using the framework to build a trusted computing component based on the reference monitor concept [13]: the Least Privilege Separation Kernel (LPSK). The third stage will be a third party evaluation of the LPSK. Finally, the results from the project will be made available. Source code, as well as, the documentation will be made available as open source [2]. The work presented here consists of building and testing a prototype that exhibits a subset of the interface functionality of the least privilege separation kernel.

C. OPEN SOURCE CONFIGURABLE OPERATING SYSTEMS

To enable the building of the prototype, a configurable operating system is used. An open source configurable operating system is an open source development environment that can be used to create, alter, and/or update kernels and their internal mechanisms. This section discusses the advantages and disadvantages of open source software and provides a list of supported features for configurable operating systems.

1. Open Source Development Advantages

Keeping a project open source has many advantages. Open source projects allow a larger amount of scrutiny among peers through openness of viewing source code and procedure documents. Keeping a project open source allows other groups to extend the research undertaken and add input. In addition, open source projects, relative to proprietary projects, often do not need as many resources expended to sell good ideas.

2. Open Source Development Disadvantages

However, open source projects also have disadvantages. Open source projects may have very little funding. Some open source companies such as Red Hat Inc. have been able to fund such projects by charging fees for technical support. Another problem

is many open source development kits have been abandoned or have had limited recent contributions, making it difficult to run the latest hardware and/or software.

3. Supported Features of Configurable Operating Systems

A configurable operating system may have many features, depending on its intended use. Certain features are advantageous for use in developing the prototype. These features include: platform support, hardware multi-ring support, memory isolation, ease of use for compilation (i.e., configure, make, and make install), I/O support, debugging support, simple design, kernel system calls, and memory management.

D. SEPARATION KERNEL

A separation kernel is a kernel that allocates resources, blocks or partitions, and mediates flow between blocks. Its mechanisms run in the most privileged domain of the system. A separation kernel has both advantages and disadvantages over the traditional security kernel approach. These advantages and disadvantages will be introduced along with the Least Privilege Separation Kernel (LPSK) approach which is slightly different than the typical separation kernel approach.

1. Separation Kernel Advantages

The separation kernel approach simplifies the information flow checking mechanisms of the kernel base by preventing direct interaction between processes that have been separated. The separation kernel approach works by partitioning system resources. This approach also allows the separation kernel to be very simple, for evaluation, and moves non-security relevant processing out of the separation kernel.

2. Separation Kernel Disadvantages

Policies such as those captured by the Bell-LaPadula [14] model are not part of the separation kernel definition or implementation. Thus when a separation kernel is used to enforce a Multi-Level Security (MLS) policy, extraordinary care must be taken to define the kernel configuration data.

3. Principle of Least Privilege approach

The principle of least privilege approach limits information flow between partitions to only the flow required to achieve the desired functionality of the system as a whole.

III. KERNEL KIT SELECTION

This chapter describes the selection process for choosing the kernel kit that was used for the building of the prototype. The eventual choice was OSKit. To understand the reason for this choice, one must review the requirements used in the kernel kit selection. A kernel kit provides the ability to customize and extend the features and capabilities of the kernel.

A. KERNEL KIT REQUIREMENTS

To enable the development of the kernel prototype, a kernel kit has to be chosen. A sample of kernel kits was taken based on two requirements: the operating system could not be too large and complex and it had to be written in the C/C++ language and/or assembly. The sample set of candidate kernel kits was not intended to be comprehensive but merely to include the most predominant small scale operating systems that appeared to provide kernel development capabilities. After the sample set was identified, the various operating systems were reviewed against requirements to determine the best choice for the needs of the prototype. The requirements were subdivided into two key requirement divisions: mandatory requirements and suggested requirements. If any mandatory requirement was not met, the operating system was removed from further consideration.

1. Mandatory Requirements

The chosen mandatory requirements were based on the following factors: cost, availability, and pertinence for the prototype. In addition, it must run on an x86 platform, provide a stable release, have available source code that can be altered, and have more than one privilege domain.

a. x86 Platform

The requirement to execute on an x86 platform is mandatory because of the necessary features it provides. These are: segmentation, privilege levels, hardware tasks, and call gates.

(1) Segmentation. Segmentation is implemented in the x86 architectures and supports highly granular management for the address space of each process. Segmentation can support process isolation and controlled sharing of objects among processes.

(2) Privilege Levels. An x86 platform supports four hardware privilege levels: 0 to 3, with 0 being the most privileged. Privilege levels permit the most privileged domain, i.e., that of the kernel, to be protected from applications. Privilege levels are totally ordered and permit tasks to be organized in ways that support least privilege.

(3) Task State Segments (TSS). In multitasking environments where tasks must concurrently run, a task's running state must be saved. A task's running state is stored in its task state segment. The other benefit of the use of a TSS is code efficiency. The context switch can happen with only one instruction. This allows context switching to be performed easily.

(4) Call Gates and Traps. When tasks are running in a less privileged domain and must access resources available in a more privileged domain, execution goes through a call gate. Hardware privilege checks ensure that services are provided to the caller while ensuring the integrity of the other privilege domains within the task.

b. Stable Release

Providing a stable release of the system from which the prototype will be built is a mandatory requirement. A stable release means that the kernel kit has been

thoroughly debugged and tested. Spending too much time trying to track down software bugs in the prototype is not a luxury the constricted development timeframe provides.

c. Open Source Code Environment

An open source code and environment requirement is mandatory because of the benefits of modification, simple acquisition, and cost.

(1) Modification. Due to the nature of the prototype being built, the kernel kit must be modifiable. No other known and available open source prototype exists for a system implementing the strong memory partitioning with fixed task scheduling intended in this project. This project will take advantage of existing drivers for network and I/O controls and basic operating system operations such as message passing and threading. In building the prototype, software reuse with modifications as needed makes the most sense. Due to these reasons, the requirement of being able to modify the source code is mandatory.

(2) Simple Acquisition. In order to modify the code, one must be able to acquire the code in an easy acquisition process. Understanding how to download, install, and run the software should not be difficult. The problems that prevent simple acquisition include proprietary code, lack of documentation, and lack of archiving support. Some Open Source Standards take care of these problems through GNU licensing agreements by following the three step Unix installation standard (i.e., configure, make, and make install), and by having online archives of source code. Use of the Unix compilation standard helps with any lack of documentation for installation. An online archive of source code helps provide the ability to download the source code in an easy manner.

(3) Virtually No Cost. Purchasing expensive software for the prototype is not an option. The benefit of open source software is that it can be acquired at minimal or no cost.

d. More Than one Privilege Domain Environment

To provide the ability of the prototype to partition memory and securely control information flow, a kernel that implements user and kernel tasks is mandatory. The requirements for having more than one privilege domain is achieved through hardware support for privilege domains, and code support running in the most privileged domain.

(1) Hardware Requirement. The hardware requirement to enforce separation of privilege domains is provided by the x86 platform. In x86 platforms, the CPU checks memory accesses and permissions and traps when a task tries to access memory not permitted.

(2) Code Support. When the hardware interrupts due to non-privileged attempts to access privileged instructions, the software must properly handle these faults. As part of the mandatory requirement to use more than one privilege domain, it is required that trap-handling mechanisms be built in and easy to modify.

2. Suggested Requirements

The suggested requirements for the system are as follows: simple code generation support, actively maintained software, and debugging support. It is not assumed that any one kernel kit will meet every mandatory and suggested requirement; however, the suggested requirements help narrow the search for the best kernel kit.

a. Simple Code Generation Support

A suggested requirement is set of simple mechanisms to build the kernel. These may include make files and other tools.

(1) Make-File Like Procedures. Due to the complexity of operating systems, compiling and linking in all of the include files and resources can be a daunting task. To remember and type in by hand all of the necessary inputs for the prototype to be built is difficult at best.

Choosing kernel kits that provide a make file mechanism solves this problem. The make file may require some modifications for the prototype.

(2) Quick Compile Times. A disadvantage of building binaries for an operating system is the time it takes to compile them. This can slow development time.

To resolve this development time problem, it is important the targeted kernel kit be comprised of a set of binaries that can be linked together. This separation speeds up compilation when large chunks of code do not need to be recompiled but simply linked in. Having make files that only recompile the changed files is helpful at speeding up development time and thus is a part of the suggested requirement for simple code generation support.

b. Actively Maintained

Another suggested requirement is that the chosen kernel kit is actively maintained. To be deemed actively maintained, the website that provides the location to acquire the source code must be recently updated by the webmaster, providing recent stable releases of the source code, and encouraging feedback for new releases.

(1) Website Maintained. To determine whether a kernel kit is actively maintained, a check is made to see if the hosting website is active. The software must be supported by an active community for active maintenance. Hardware devices constantly change and update. Requiring an operating system that can work with and support each new device is important. An indicator of active maintenance is having recent software releases.

(2) Recent Updates to Software. Recent software updates is part of the suggested requirement for active maintenance. Debugging software that is no longer maintained will slow down development of the prototype. If the software is no longer actively maintained, the kernel kit must have a stable release.

(3) Encourage Feedback. A community that encourages feedback with the use of the operating system or kernel kit is a nice part of active maintenance.

c. Debugging Support

A suggested helpful requirement is that the targeted operating system or kernel kit will provide means to debug the prototype. It may not be feasible to have a high level rich debugging environment for debugging the prototype. Providing a GDB stub with debugging hooks in the code to step through the executing code is enough to meet the suggested requirement.

B. KERNEL KIT CONSIDERED

The kernel kits considered are listed with their various properties relevant to the selection process (See Table 1). The kernel kits that have a 'No' tag associated with a mandatory requirement are eliminated from further consideration.

Kernel Kit	Developers	Link	X86	Stable Release	Open Source	Privilege Domains	Simple Code Generation	Active Maintanence	Debug Support
Choices	University of Illinois	http://choices.cs.uiuc.edu	Yes	Yes	Yes	Yes	Yes	Yes	Yes
ECos	Cygnus, Red Hat, ECosCentric	http://ecos.sourceware.org/getstart.html	Yes	Yes	Yes	Yes	Yes	No	Yes
OSKit	University of Utah	http://www.cs.utah.edu/flux/oskit	Yes	Yes	Yes	Yes	Yes	No	Yes
Fiasco	TUD-Dresden, University of Technology	http://os.inf.tu-dresden.de/fiasco	Yes	Yes	Yes	Yes	Yes	Yes	Yes
K42	IBM	http://domino.research.ibm.com/comm/research_projects.nsf/pages/k42.index.html	Yes	No	Yes	Yes	Yes	Yes	Yes
Pebble	Bell Labs	http://www.	No	Yes	Yes	Yes	Yes	Yes	Yes

		bell-labs.com/project/pebble							
Spin	University of Washington	http://www.cs.washington.edu/research/projects/spin/www	Yes	Yes	Yes	No	Yes	No	Yes
TinyOS	University of Washington	http://www.cs.washington.edu/research/projects/spin/www	Yes	Yes	Yes	No	Yes	No	Yes
MMLite	Microsoft	http://research.microsoft.com/invisible	Yes	Yes	Yes	No	Yes	Yes	Yes

Table 1. Reviewed Kernel Kits.

C. FINAL DETERMINATION

After eliminating from the sample those operating systems or kernel kits that did not satisfactorily meet the mandatory and suggested requirements, two kernel kits were chosen for deeper consideration and analysis. An analysis of the deciding factors and reasons for selection are given.

1. Final Deciding Factors

Eliminating most of the operating systems due to the previously stated reasons in Table 1 narrowed the choices to the following four systems: Fiasco, Choices, ECos, and OSKit.

a. Fiasco

Fiasco is developed at TU Dresden. It is compatible with the x86 L4 microkernel. It is a real-time preemptive kernel written in C++. Fiasco's adherence to the specification of the L4 microkernel makes it a desirable choice. The downside is Fiasco is too specialized for what is needed by the prototype.

b. Choices

Upon initial review, Choices did not appear to have a stable release that ran on the tested x86 platform. Initially, the Choices operating system would not install. Debug support was not provided with Choices to remedy the situation. Due to acknowledged author error, Choices was eliminated prematurely. FiSh, a shell application, is included with Choices to run applications using the Choices operating system, making it a desirable choice. Choices is designed for systems research at the university level.

c. ECos

ECos was developed in 1997 by Cygnus Solutions after success supporting GNU GCC and GNU GDB. Cygnus Solutions built a real time operating system for their GNUPro tool suite. ECos is a highly configurable embedded systems kernel built for many different types of architectures and platforms.

The benefits of ECos are the following: ECos configurable tools are available in a GUI environment; it supports command line execution, memory pools for fixed sized memory allocation are used, remote GDB debugging is supported, and it has plenty of documentation.

The disadvantages of ECos are the following: no memory protection is implemented, the latest stable release was in May 2003, and it lacks non-preemptive schedulers.

d. OSKit

OSKit was developed at the University of Utah to provide a platform to lessen the cost of doing operating system research and development.

The benefits of OSKit are the following: modular design, brevity of code, remote GDB debugging support, a simple process library, online documentation, non-preemptive scheduling, and adequate sample code for the various kernel features (threading, multitasking, timers, etc.).

The disadvantages of OSKit are the following: the x86 LDT is not used, and the kernel kit developers released the last stable update in March 2002.

2. Final Selection

Of the four kernel kits, OSKit and ECos were selected for more critical review. Fiasco was too specialized for what was needed by the prototype and therefore was not considered. Choosing between OSKit and ECos was difficult. Not surprisingly, ECos provided GNU GDB debugging support. OSKit also provided the ability to add GDB stub controls for the GDB debugger to work remotely. Both operating systems are built to be modular and highly configurable. ECos and OSKit provide reasonable documentation, installation ease, and make file support. Neither had recent stable releases, forcing any development to be done on older hardware and operating systems. For enabling development, ECos exceeded expectations by providing command line execution of the created binaries to speed up development time. ECos also appeared more able to meet the needs of the prototype with regard to static memory allocation and memory pools. However, coding in OSKit was easier, and the kernel is more developed for kernel and user space separation.

In the end, the simplicity of OSKit made it the better choice. It has few dependencies between components. Unlike ECos it has existing libraries for multitasking of processes to support kernel and user tasks. In the final analysis, OSKit proved the better choice, as static non-preemptive scheduling and memory partitioning for multi-tasked processes are highly attractive, and OSKit seems to provide the simplest platform to enable these. An implementation was built using OSKit to provide memory partitioning and the sharing of resources among processes.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. PROTOTYPE IMPLEMENTATION

The LPSK prototype is built by enhancing the configurable operating system, OSKit. OSKit separates memory between the kernel space and user space through paging. User processes have separate address spaces. In OSKit, data can be transfered between user processes using memory resources managed by the memory manager. The prototype enables the system to govern rules established by the configuration file of process and resource interaction. The implementation allocates memory into partitions and memory resources within partitions. The altered kernel mediates the memory sharing with paging and page fault trap handling. This enhanced OSKit enables the separation of memory as defined by the configuration file. The steps to enhance the kernel are as follows: setting up the development environment, using version control, implementing the configuration file, partitioning memory, creating and allocating resources, and altering user processes.

A. SETTING UP THE DEVELOPMENT ENVIRONMENT

Setting up the development environment is necessary to build the prototype. Time was spent finding the most suitable configurable operating system. Also, time was allocated to become familiar with OSKit to determine the least amount of changes needed for the required enhancements. Having a working development environment enables the ability to become familiar with OSKit and build the prototype. Building the development environment requires the following procedures: acquiring hardware, choosing a compatible software platform, installing OSKit, and implementing version control. A more detailed explanation of setting up the development environment can be found in Appendix A.

1. Hardware

Building the prototype using OSKit requires certain hardware. An Intel PC with 32MB of RAM and a 400MHZ processor and an old TNT graphics card will permit the prototype to run. A machine with one GB or more of RAM will not permit the prototype to run without modifications to OSKit.

2. Software Compatibility

OSKit will not compile in all software environments. The last stable release of OSKit was in 2002. Consequently, OSKit will only compile with certain tools and versions. For OSKit to compile, it requires: "GNU make, GNU CC version 2.7.x or version 2.95.2, and GNU binutils version 2.8.x, or 2.9.1 with BFD 2.9.1" [15]. Once a system is configured correctly with the compatible software required, OSKit can be compiled and the prototype built.

3. Building the Prototype

Building the prototype for the target machine involves compiling OSKit and linking with the code for the prototype. Additionally, the prototype requires the Simple Process Library (SPROC) that comes packaged with OSKit in order to add multi-process capabilities to the prototype.

a. Compiling

Compiling OSKit involves a multi-step process. The first step to compilation is to download the March 2002 release of OSKit into a directory on the target machine. Next, one must execute the file named 'configure', type 'make', and finally 'make install'. The target machine now has the environment variables and the source code compiled to run test kernels. These kernels must be linked together to include all of the necessary kernel building blocks. One such test kernel is the prototype.

b. Linking Together the Kernel Image

After the source code is compiled, the prototype must be linked. The 'sproc' directory under 'oskit/examples/x86/' has example code for multi-booting a kernel with user processes that use the Simple Process Library. The multi-boot program called 'mkmb2' links with the kernel object, configuration file, swap file, and user process objects. Running the multi-boot program correctly generates the prototype image file. Now, the prototype is ready for changes. To make needed changes to the prototype, version control is important.

4. Using Version Control

Version control provides a way to manage software updates, preventing loss of data and allowing revision to previous software versions. For the project, the version control software Subversion was chosen.

Subversion manages system changes. It creates system backups by saving code for each new release automatically during code updates. For the project, a Subversion server was set up. A repository was created with a clean copy of OSKit. Subsequent changes to OSKit are controlled through Subversion by monitoring check in and check out of the software.

B. IMPLEMENTING CONFIGURATION FILE

A requirement of the prototype is to allocate memory based on specifications in a kernel configuration file. The configuration file must be read into kernel space before any allocation of resources or creation of user processes. In the prototype this is accomplished by linking the configuration file to the kernel image and reading the configuration after booting the kernel image.

1. Configuration Data

The configuration file as implemented includes data to define the following: partitions, processes, resources, partition flows, and resource flows. These are implemented and handled in the prototype.

a. Partitions

Partitions are abstract entities to which resources are allocated. Resources include both active and passive entities as well as execution time.

b. Processes

Processes have priority assignments used for scheduling within a partition and fixed assignments used for static scheduling of partition time slices. Processes can act as both subjects and objects. They act as subjects when they act on objects such as resources. In addition, they act as objects when acted upon by other processes. The access by processes to resources is mediated by flow rules enforced by the separation kernel.

c. Resources

This prototype deals exclusively with memory resources. Other types of resources will be implemented in the future versions of the LPSK prototype.

d. Partition Flows

Partition Flows are directional flows that have a designated source partition and a designated destination partition. The maximum number of partition flows in a system is the number of partitions in the system multiplied by one less than the number of partitions in the system. The access modes possible in partition flows are the following: read, write, and read/write.

In the prototype, every access mode is addressed. Execute reads and execute writes are not permitted. An execute read is an instruction that reads the current

instruction pointer. An execute write is an instruction that writes to the current instruction pointer. Page faults due to attempted execute reads or writes will terminate the process. An execute read or write is known to be attempted when the instruction pointer and page fault address in the *cr2* register contain the same value. A read happens when a process stores the contents of memory into a register or memory. A write occurs whenever contents in memory or in a register are written to a memory location.

e. Resources Flows

Resource flows are directional flows that have an association between a resource and a process with a defined access mode. The association depends on the access mode. The access modes allowed are read, write, and read/write. The maximum number of resource flows in a system is the number of resources in the system multiplied by the number of processes in the system.

In the prototype, a memory access cannot occur unless there is a defined partition flow and resource flow with the access privileges to permit the access. If any one of the two flows does not exist, the process is terminated.

2. Porting the Configuration File to the Prototype

The prototype reads the configuration file to obtain the permitted flows, create partitions, allocate resources, and manage processes. The configuration data is linked into the kernel prototype image.

a. The Application to Construct the Configuration File

An application called configapp was built to write the configuration data directly to a file in binary form. The source of this application appears in Appendix C This application uses the same compiler and the same configuration header file as the prototype. The data structure LPSKconfig is written to file by configapp and is the same data structure used to retrieve the data by the prototype during execution. There are key reasons this application is using the same compiler and defined data structure from the configuration header file as the prototype.

First, the same GCC compiler is used for compiling both the prototype and configapp. This is to prevent any issues of different data types having different sizes.

Second, the LPSKconfig structure is reused, allowing configuration data to be written and read properly in the correct order and with the correct size. This is the same structure used by both the prototype to store the configuration file data and by configapp.

b. Binary Reading of the Configuration File

To read the binary information in the configuration file, the prototype allocates memory for the LPSKconfig structure and then reads the binary into the structure. Since the configuration file is linked into the kernel image, the file has to be retrieved using the procedure provided by OSKit, 'oskit_absio_read'.

C. PARTITIONING OF MEMORY

The prototype has the ability to separate memory locations into partitions access to which is mediated by the kernel in accordance with the configuration. Partitioning results from separating memory through paging, implementing a page fault trap handler, and defining partition flows.

1. Partitioning via Paging

Paging provides separation of memory between partitions. When a user-domain process is created, a check is made to determine to which partition the process is allocated. The minimum and maximum memory addresses of the partition are applied to the user-domain process. The prototype maps the pages to physical memory based on these addresses for use by the user-domain process. After the pages are mapped for the process, the process is initialized and started in user mode. The mapping is intentional. Any attempt by the user-domain process to access pages that are not mapped to its partition results in a hardware page fault.

a. Hardware Page Faults

Whenever there is an access to a page not mapped within physical memory, hardware sends an interrupt, which results in a page fault. A page fault trap handler was constructed for the prototype to handle the page fault interrupts sent by hardware. The page fault handler built into the prototype does not handle all Intel x86 opcodes that may cause page faults. Instead it demonstrates that page faults may be handled. All major types of memory access handling are addressed by the page fault handler, which are termed as memory access classes. These memory access classes are: constant to memory, memory address to register, memory pointed by a register to register, register to memory pointed by a register, incrementing or decrementing memory, pushing or popping a memory address, memory to memory, memory to register, register to memory, pushing memory, popping memory, and accessing program counter address. Also handled are the following memory access modes: read, and write. A read execute or write execute page fault will terminate the process.

2. Page Fault Trap Handler

The prototype's page fault trap handler reads the line of code that caused the fault, checks flows to determine access permissions, handles the line of code, and returns control to the next line of code following the line of code that caused the page fault. The trap handler terminates any process that performs an illegal access.

a. Handling Paging

A kernel data structure describes the address space of each partition/process in terms of pages. If a page fault occurs, this database is referenced. If the page is in the address space to which the process has permission for the access mode attempted, then the kernel executes the instruction that initiated the memory access and returns control following the instruction, otherwise the process is terminated.

b. Handling of Opcodes

Before handling the requested access, the memory access attempt must be determined. The opcode that caused the page fault determines the memory access attempt. If the memory access is deemed legal, the opcode is handled in the kernel-domain within the trap handler. The opcode is handled by editing the stored state of the processor when the trap occurred, and this stored state becomes the state of the processor when control returns back to the process in the user-domain. The ability to read and write the state of the processor when the trap occurred allows the handling of all page fault traps used for the purposes of the prototype.

The prototype has a case statement that handles two-byte value inputs with a case statement in the default section that handles one-byte value inputs. This is done to efficiently handle the opcodes that may be one or two bytes in length. Any opcode unhandled will cause the termination of the process. After a successful operation is performed, the mapping of the required page will be removed and the original mapping restored. Once this mapping has been performed, execution can be returned to the user-domain process. Before this operation can occur, permissions are checked to determine that the access attempt mode is allowed.

3. Permission Handling

The page fault trap handler checks the permissions allowed. These permissions are between the source and destination partitions of the memory attempt and the resource being accessed. Retrieving the process ID determines origin of the page fault. Using this ID, the partition flows and resource flows that exist for this process are obtained and the access modes allowed are read. A required partition flow must exist to permit the access. If the access is permitted, another check is made to determine that a resource flow exists with the required access permission. If both flows exist, access to the page is granted.

a. Returning to User Process

To return execution back to the user process, the prototype must prevent the same page fault from occurring again. An infinite loop will occur if execution is returned without advancing the instruction pointer; the process will execute the same operation that caused the page fault. To solve this problem, the instruction pointer must be updated to skip past the instruction that caused the page fault. These updated values are based on the instruction set of the hardware. For example, a four byte instruction is handled by incrementing the instruction pointer by four. Upon performing this operation, execution can return back to the process, and successful handling of memory partitions and resources is complete.

The enhanced changes made to OSKit to build the LPSK prototype enforce memory separation into partitions and the ability of processes to access data resources as defined by the configuration file. These enhanced changes made to the original OSKit release of snapshot 20020317 are detailed in Appendix C.

THIS PAGE INTENTIONALLY LEFT BLANK

V. PROTOTYPE TESTING AND RESULTS

This chapter deals with the functional testing of the prototype and the results. The prototype has the ability to separate memory locations into partitions and resources that the kernel mediates. Partitioning is the result from separating memory through paging, implementing a page fault trap handler, and defining partition flows. Resource separation results from defining resource flows that are checked in the page fault trap handler when a process in another partition attempts access. This prototype is tested to ensure that only the correct memory based accesses occur between partitions and resources. These are specified by the policy encoded in the configuration file.

Individual prototype tests are conducted using a development machine and a testing machine. The development machine runs the prototype that has a GNU Project Debugger (GDB) stub to permit remote debugging from the testing machine. The testing machine runs the GDB debugger remotely connecting to the GDB stub via a serial link.

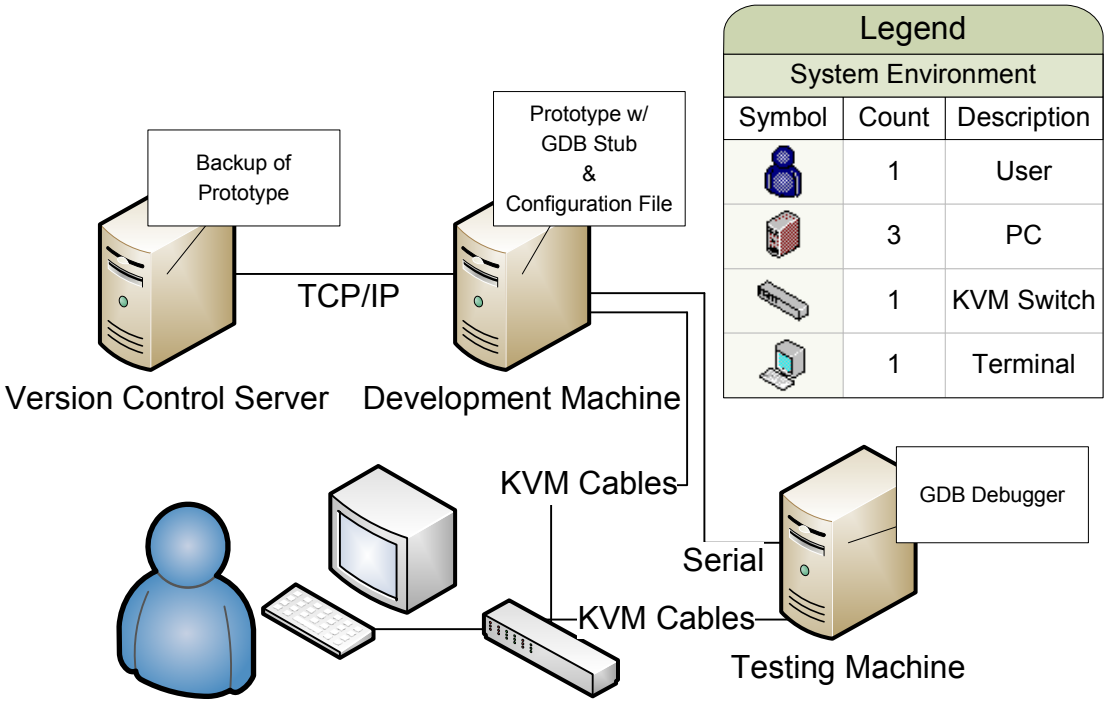

Figure 1. System Topology.

The subsequent section describes the following: test plan, setup, functional testing, the results, and any problems encountered.

A. TEST PLAN

Tests are conducted on all memory access classes ensuring the access modes of read, write, read/write, and execute are tested systematically. Memory access classes discussed here are memory accesses that are possible in x86 architectures without specifying a specific register, address, or value. The memory access classes are as follows: constant to memory, memory address to register, memory pointed by a register to register, register to memory pointed by a register, incrementing or decrementing memory, pushing or popping a memory address, memory to memory, memory to register, register to memory, pushing memory, popping memory, and accessing program counter address.

Each test results in a success or a failure. The results are annotated with a success or failure tag in Table 3. A success result occurs when the memory access class tested is properly handled via the prototype's trap handling. A failure happens when the memory access class is not handled via the prototype's trap handling.

B. FUNCTIONAL TESTING

Performing the functional tests on the prototype requires setting up an environment based on policy which is derived from the configuration file instantiation. The testing environment requires certain characteristics to be initialized in the following: processes, partitions, resources, partition flows, and resource flows (see Figure 2). To perform the functional tests, various values in the resources are accessed in different modes and the subsequent state is inspected.

1. Address Values

The address values accessed in the tests are the virtual addresses '0xa0009fac' and '0xa0009fa8'. These are the resource locations of two resources in Partition 2. The

tests involve changing the initial values, instructions, and access modes and verifying that the correct values reside in the correct memory locations.

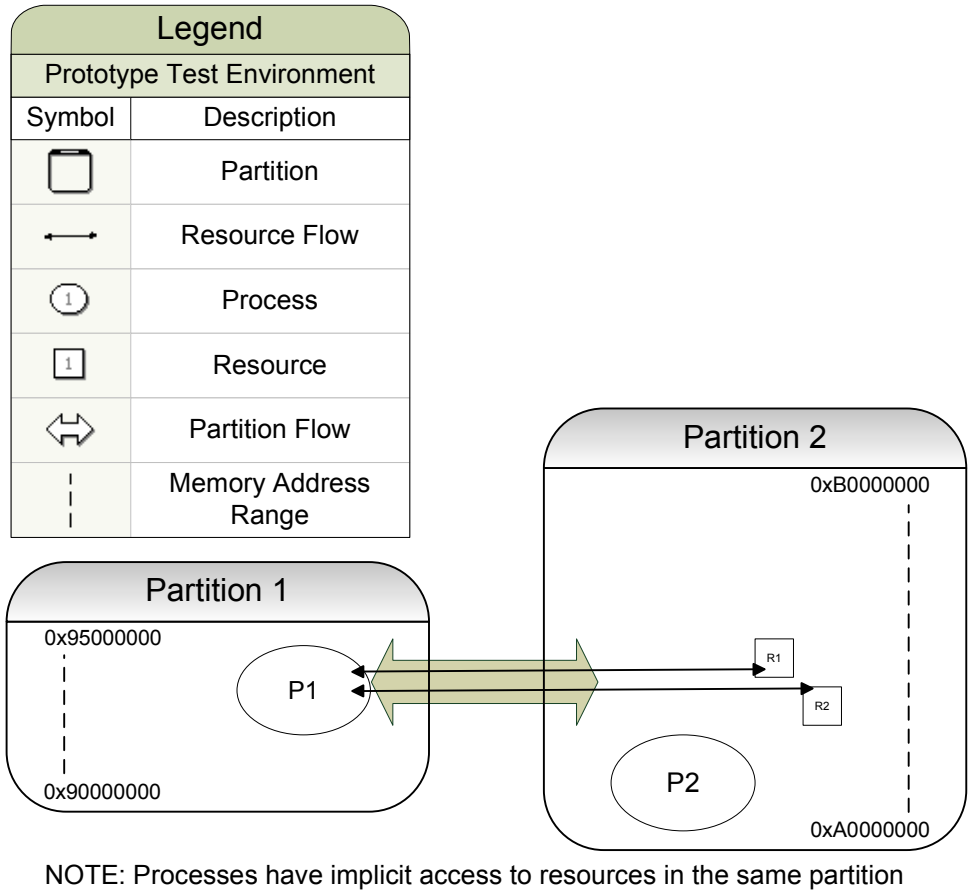

Figure 2. Testing Topology.

2. Test Cases

Test cases verify that the various accesses to memory are constrained correctly. These test cases consist of all the opcodes handled by the prototype. The AC8 test case is a success when the memory access attempt results in the termination of the user process that made the attempt. The other test cases are considered a success when a memory access attempt by an instruction running in user mode causes a page fault and is properly handled by updating the system state (or not) according to policy, then returning back to the user process at the next instruction. These tests cases consist of the tests described in Table 2.

Test	Access	Type	Description
AC1	Write	Constant to Memory	Test AC1 certifies the prototype handles memory accesses due to copying a constant value into a memory location.
AC2	None	Memory Address to Register	Test AC2 certifies the prototype handles memory accesses due to copying a memory address into a register.
AC3	Read	Memory Pointed by a Register to Register	Test AC3 certifies the prototype handles memory accesses due to copying memory referenced by a register into a register.
AC4	Write	Register to Memory Pointed by a Register	Test AC4 certifies the prototype handles memory accesses due to copying values in a register into memory referenced by a register.
AC5	Write	Incrementing or Decrementing Memory	Test AC5 certifies the prototype handles memory accesses due to incrementing or decrementing.
AC6	None	Pushing or Popping Memory Address	Test AC6 certifies the prototype handles memory accesses due to pushing or popping a memory address onto or off of the stack.
AC7	Read & Write	Memory to Memory	Test AC7 certifies the prototype handles memory accesses due to copying directly memory from one location to another.
AC8	Read	Memory to Register	Test AC8 certifies the prototype handles memory accesses due to copying memory into a register.
AC9	Write	Register to Memory	Test AC9 certifies the prototype handles memory accesses due to copying a register's contents to memory.
AC10	Read	Pushing Memory	Test AC10 certifies the prototype handles memory accesses due to pushing memory onto the stack.
AC11	Write	Popping Memory	Test AC11 certifies the prototype handles memory accesses due to popping values off of the stack into memory.
AC12	Execute	Accessing Program Counter Address	Test AC12 certifies the prototype handles memory accesses due to trying to access the instruction pointer address.

Table 2. Memory Access Class Test Descriptions.

C. RESULTS

The functional tests were performed against the prototype instance detailed in Figure 2, and the results were collected. The data collected was checked to verify that the prototype functions as intended. The results of the successes and failures of the test cases run are displayed in Table 3.

Test	Access	Type	Expected Result	Actual Result
AC1	Write	Constant to Memory	Success	Success
AC2	None	Memory Address to Register	Failure	Failure
AC3	Read	Memory Pointed by a Register to Register	Success	Success
AC4	Write	Register to Memory Pointed by a Register	Success	Success
AC5	Write	Incrementing or Decrementing Memory	Success	Success
AC6	None	Pushing or Popping Memory Address	Failure	Failure
AC7	Read & Write	Memory to Memory	Success	Success
AC8	Read	Memory to Register	Success	Success
AC9	Write	Register to Memory	Success	Success
AC10	Read	Pushing Memory	Success	Success
AC11	Write	Popping Memory	Success	Success
AC12	Execute	Accessing Program Counter Address	Success	Success

Table 3. Test Results.

As can be seen in Table 3, the test results demonstrated that the prototype did in fact handle all of the memory access classes as expected.

THIS PAGE INTENTIONALLY LEFT BLANK

VI. CONCLUSION AND FUTURE WORK

This chapter discusses the conclusions and future work of the LPSK prototype to supplement the TCX Project.

A. CONCLUSION

The goal of this thesis was to construct a prototype of certain aspects of the functionality for a high assurance separation kernel that reflects the Principle of Least Privilege (PoLP) [1] and that will enable the TCX Project. By implementing partitioning of memory and monitoring memory resources within the separation kernel prototype, the prototype allows the Principle of Least Privilege to be realized for memory resources of the LPSK prototype. The tests described in Chapter V show that the approach taken to monitor memory accesses between partitions is feasible and may be used in the TCX Project to protect memory.

The approach taken of using paging to prevent unwarranted accesses as described in this thesis enforces rules mandated by the configuration file in a straightforward manner. Also, this approach prevents processes from accessing the instruction pointer to point to or read locations in other partitions.

Implementation of this prototype leads to ideas for future work that would be of benefit to the TCX Project.

B. FUTURE WORK

During the development of this thesis work, four key areas were found to warrant future work.

1. Complete Handling of Opcodes in the Page Trap Handler

Complete handling of all of the x86 opcodes that may cause page faults is not fully implemented in the prototype's page trap handler. The proof of concept work is

done to show that all of the x86 opcodes can be implemented. Further work in this area would be to implement the complete set.

2. Static Scheduling

The prototype reads the configuration file to determine the number of scheduled ticks for each process and sends this to the scheduler. The scheduler, while sufficient for this prototype, is not fully implemented. Further work may be done in this area.

3. Handling Other Resource Types

This work implements memory resource handling in the LPSK prototype. Further work is necessary to handle other resource types.

4. Efficient Caching of Resource Accesses

The efficiency of handling memory accesses outside of a process' partition can be improved by building into the prototype caching mechanisms to minimize the number of page faults. Implementing a caching mechanism could benefit the work done by providing the benefits this approach offers while minimizing the performance penalties.

APPENDIX A: INSTALLATION ACTIVITIES

A. ENIVRONMENT INSTALLATION

The procedures taken to install the working environment are as follows: connecting computers via serial link, installing Linux Red Hat 7.2, installing GRUB, and connecting to a Subversion server.

1. Serial Link Connection

To debug the prototype remotely with a GDB debugger, the selected test machine and development machine are connected via a serial line. This serial line is connected via serial ports and a cable. The cable used is a serial crossover cable to permit proper communication.

2. Linux Installation

All of the procedures in this thesis are run with Linux Red Hat 7.2 installed on both machines. After installing and running Linux out of the box on the both machines, GRUB needs to be installed on the selected development machine. Linux Red Hat 7.2 was chosen for ease of use with OSKit.

3. GRUB Installation

Installing the GRUB package can be done following the Red Hat documentation to install GRUB since it will not be installed via the Red Hat Linux installation process. After the GRUB package is loaded, one must open a root shell command prompt and run the command, '/sbin/grub-install <partition>', where '<partition>' is the location of the first primary partition. This will install the GRUB stage 1 boot loader, "to the MBR of the master IDE device and on the primary IDE bus". After running these procedures to install GRUB, the GRUB graphical loader will display on boot [16].

After GRUB is installed, the 'grub.conf' file is added and should be edited to include the prototype image file that will be run. The way this is done is by copying the

title, root, and kernel lines of the Linux parts in the 'grub.conf' file and editing the copied lines with the title line changed to describe the LPSK prototype and the kernel line changed to specify the location of the prototype image.

4. Connecting to Subversion Server

To install the SVN client in the Linux environment, Apache Runtime (APR) as well as Subversion has to be installed. SVN with APR can be set up in many different ways, running the following commands is one way:

```
cd /
wget http://subversion.tigris.org/downloads/subversion-1.4.4.tar.gz
tar –zxf subversion-1.4.4.tar.gz
cd subversion-1.4.4
svn co http://svn.apache.org/repos/asf/apr/apr/branches/0.9.x apr
cd apr
./buildconf
./configure
make
cd ..
svn co http://svn.apache.org/repos/asf/apr/apr-util/branches/0.9.x apr-util
cd apr-util
./buildconf
./configure –with-apr=/subversion-1.4.4/apr
make
cd ..
./configure
make
make install
```

A Subversion repository was set up on the version control server. To populate the repository a Subversion client is installed and the "svn import" command is issued. The original OSKit files were imported into the repository using this procedure. Checking in and checking out files allows OSKit to be modified and the LPSK prototype built. To check-in and check-out existing files of the OSKit repository, the following commands are used, where '<username>' represents the username, '<password>' represents the password, '<message>' represents the message to save, and '<repository location>' represents the location of the repository:

```
svn ci –username <username> --password <password> -m "<message>"
svn co <repository location> –username <username> --password <password>
```

Any additional files that are made during the development of the LPSK prototype are added by issuing the 'svn add <filename>' command, where '<filename>' represents the file to be added to the repository [17]-[19].

B. OSKIT INSTALLATION

Installing OSKit in the environment described Section A of this Appendix involves downloading OSKit's "St. Patrick's Day" snapshot from the Flux website, http://www.cs.utah.edu/flux/oskit/software.html, untaring, and installing. Complete the following steps and OSKit and the kernel image that will become the prototype will be compiled and ready for modification:

```
mkdir /oskit-dev
cd /oskit-dev
wget ftp://flux.cs.utah.edu/flux/oskit/oskit-20020317.tar.gz
tar –zxf oskit-20020317.tar.gz
cd oskit-20020317
./configure
make
make install
cd examples/x86/sproc
make Image
mkmb2 kernel swapfile usermain_testsproc usermain_hello
```

OSKit has a variety of sample kernels that can be created in different ways. See the OSKit documentation for more information [20].

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B: TEST PROCEDURES

The various tests run against the memory access classes and their results are discussed in Chapter V. A memory access class is a memory access performed by an instruction using a nonspecific register, address, or value. The memory access classes are as follows: constant to memory, memory address to register, memory pointed by a register to register, register to memory pointed by a register, incrementing or decrementing memory, pushing or popping a memory address, memory to memory, memory to register, register to memory, pushing memory, popping memory, and accessing program counter. The technical details of how these tests were performed against the memory access classes are described in this Appendix. The relative source code not listed in this Appendix can be found in Appendix C.

OSKit has the ability to provide GNU debugger (GDB) stub routines which can be called by calling various OSKit included routines. The LPSK prototype enables GDB debugging upon initialization with the following code that comes with OSKit:

```
extern struct termios base_raw_termios,base_cooked_termios;
printf("setting serial port for gdb\n");
base_raw_termios.c_ispeed = B38400;
base_raw_termios.c_ospeed = B38400;
base_cooked_termios.c_ispeed = B38400;
base_cooked_termios.c_ospeed = B38400;
gdb_pc_com_init(1, &base_raw_termios);

gdb_trap_mask = (1 <<T_PAGE_FAULT) | (1 << T_NO_FPU);
printf("break!\n");
gdb_breakpoint();
```

Testing the various memory access classes was performed by creating two user-domain processes with the source code detailed in the files ‘usermain_testsproc.c’ and ‘usermain_hello.c’. The ‘usermain_testsproc.c’ file contains the source code that generates the opcodes that are used to test the various access classes. The ‘usermain_hello.c’ file contains the source code to display the data contained in the resources that are accessed by the memory accesses. The memory accesses are performed by the code found in the ‘usermain_testsproc.c’ file.

The GNU application objdump was used to view the outputted assembly and machine code of the 'usermain_testsproc.o' object file. The 'objdump -D <object file><dump file>' command was used to dump the assembly and machine code of the '<object file>' file into the '<dump file>' file. This was done to verify that the C code written in the user-domain processes source code produces the correct x86 opcodes that are implemented in the LPSK prototype.

A. TESTING ACCESS CLASSES

All access classes detailed in Chapter V were tested. All the tests conducted ensure: the instruction executes in user mode, causes a page fault, and is properly handled by returning back to the original user process at the next instruction. The instructions used to test each access class are detailed in this section.

1. Constant to Memory

The two-byte opcode '0x05C7' tests the accessibility of storing a value to memory. The 'movl $0x64, 0xA0009fA8' instruction produces this opcode.

2. Memory Address to Register

The one-byte opcode '0xB8' tests the accessibility of storing a memory address to a register. The 'mov 0xA00DBEEF, %eax' instruction generates the one-byte opcode.

3. Memory Pointed by a Register to Register

The ability of storing memory pointed by a register into a register is tested with the two-byte opcode '0x008B' via the 'mov (%eax), %eax' instruction.

4. Register to Memory Pointed by a Register

The two-byte opcode ‘0x0289’ via the ‘mov %eax, (%edx)’ instruction tests register to memory pointed by a register access. This memory access copies the register value to memory a register points to.

5. Incrementing/Decrementing Memory

The two-byte opcodes ‘0x0583’ and ‘0x05FF’ (which were the ‘addl 0x04, 0xA0009FAC’ and ‘incl 0xA0009FA8’ instructions) tests the ability to access incrementing or decrementing memory.

6. Pushing/Popping Memory Address

The single-byte opcode ‘0x68’ is generated using the ‘push $0xA000BEEF’ instruction and tests the ability to access pushing a memory address onto the stack. The single-byte opcode ‘0x58’, is generated using the “pop %eax” instruction and follows the ‘push $0xA000BEEF’ instruction testing the ability to pop a memory address off of the stack.

7. Memory to Memory

The access type that involves copying memory to memory is tested with the two-byte opcode ‘0xA4f3’. This opcode is the ‘repz movsb %ds:(%esi), %es:(%edi)’ instruction. The opcode was realized using the code statement ‘memmove(0x0A0009FA8, 0xA0009FAC, 1)’ in C.

8. Memory to Register

Two opcodes: the two-byte ‘0x158B’ and one-byte ‘0xA1’, test the accesses involving storing memory into a register. The ‘mov 0xA0009FAC, %edx’ instruction generates the two-byte opcode and the ‘mov 0xA0009FA8, %eax’ instruction generates the one-byte opcode.

9. Register to Memory

The access type that involves copying register to memory is tested with the one-byte opcode ‘0xa3’. The ‘mov %eax, 0xA0009FA8’ instruction generates this one-byte opcode.

10. Pushing Memory

The access of pushing memory is tested with the two-byte opcode ‘0x35FF’. The ‘pushl 0xA0009FA8’ instruction generates this two-byte opcode.

11. Popping Memory

The access type that involves popping memory is tested with the two-byte opcode ‘0x058F’. The ‘popl 0xA0009FAC’ instruction generates this two-byte opcode.

12. Accessing Program Counter Address

The ‘jmp 0xA000A072’ instruction tests the access of the instruction pointer. The instruction always returns with both the *cr2* register and the *eip* register in the saved state containing the same value. The instruction runs in user mode, causing a page fault, and terminates properly in the trap handler, thus killing the user process.

B. TESTING THE PAGE FAULT TRAP HANDLER

During the testing of the various access classes, checks were made to verify that the page faults issued were properly handled in the page fault trap handler. The tests were done via GDB with various ‘assert’ and ‘printf’ statements. The code executed on every page fault when entering the page fault handler is as follows:

```
printf("handler proc 0x8%x !\n",(int)sthread->st_process);
printf("eip address: %x \n", ts->eip);
if(!from_user){//if kernel page fault
printf("kernel\n"); //or double fault then kill process
  return 1; //Don't allow double faults! So kill process
}
printf("ERR!: thread = %d, process = %p, signo = %d, "
 "code = %d, frame = %p\n", (int)pthread_self(),
```

```
sthread->st_process, signo, code, ts);
printf("page fault address 0x%x\n", cr2);
printf("eip address: %x value: %x \n", ts->eip,
*(long*)(ts->eip));
/* if page fault from accessing kernel memory, kill proc*/
if(cr2 < OSKIT_UVM_MINUSER_ADDRESS) return 1;
                                                    //sanity check
if((i = getVMSpace(cr2)) == -1) return 1;//no partition
                                                     //so kill process
```

The process was checked to verify that the process pid was the correct pid expected from the process that cause the page fault. Double faults and nested traps were detected. The *eip* register was checked by stepping through code and making sure the user-domain process' *eip* address of the instruction that caused the page fault was the same that was printed in the page fault handler. It was also checked to make sure that every page fault handled in the trap handler was handled properly with the correct *eip* address returned pointing to the next instruction of the instruction that caused the page fault. Every x86 opcode that the LPSK prototype handles was checked and verified.

The page fault trap handler was tested against every memory access class by reading and writing various values to the defined resources. The code used to test the handling of the access classes by the prototype are listed in Appendix C in the files, 'usermain_testsproc.c' and 'kernel.c', with the binary code and assembly code of 'usermain_testsproc.c' and a description of which test is performed by which line of assembly is listed in the file named 'dump'.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX C: PROTOTYPE CODE IMPLEMENTATION

The implementation of the Least Privilege Separation Kernel (LPSK) prototype consists of downloading, installing, and modifying the OSKit snapshot version 20020317 as described in Appendix A. The files added or modified to the existing OSKit are listed in this Appendix with the source for review. The files are listed as indexed from the root location where OSKit is installed. If the instructions given in Appendix A were followed with strict adherence, the root development location is '/oskit-dev/'.

A. OSKIT/SPROC.H

The 'oskit/sproc.h' file was modified with the only change being done to the 'oskit_sproc_create' function prototype.

```
/* create a process and assign a process description */
/* added min & max for residing processes in partitions */
oskit_error_t   oskit_sproc_create(const struct
                        oskit_sproc_desc *desc,
                  oskit_size_t size, oskit_size_t min,
                  oskit_size_t max, //DWC 4/5/2007
                  struct oskit_sproc *outproc);
```

B. OSKIT/UVM.H

The 'oskit/uvm.h' file was modified with the only change being done to the 'oskit_uvm_create' function prototype.

```
/* create a vmspace */ //DWC 4/5/2007
/* added min and max for creating vmspaces with fixed
   min and max constraints for partitioning */
oskit_error_t   oskit_uvm_create(oskit_size_t size,
                   oskit_size_t min, oskit_size_t max,
                   oskit_vmspace_t *out_vm);
```

C. THREADS/SCHED_POSIX/SCHED_POSIX.C

The 'threads/sched_posix/sched_posix.c' file was modified to add ticks to round robin scheduler.

```
/* added ticks to processes for round robin scheduler */
Line 260:pthread->ticks = pthread->interval_ticks;
/* added ticks to processes for round robin scheduler */
Line 433:pthread->ticks = pthread->interval_ticks;
```

D. THREADS/PTHREAD_CREATE.C

The ‘threads/pthread_create.c’ file was modified to add ticks to round robin scheduler.

```
/* added ticks to processes for round robin scheduler */
Line 199:pthread->interval_ticks=attr->sched_ticks;
```

E. UVM/SPROC/SPROC.C

The ‘uvm/sproc/sproc.c’ file was modified with the only change being done to the ‘oskit_sproc_create’ function to permit partitioning of memory.

```
extern oskit_error_t
oskit_sproc_create(const struct oskit_sproc_desc *desc,
oskit_size_t size, oskit_size_t min, oskit_size_t max,
struct oskit_sproc *outproc)
//DWC 4/5/2007 added min and max to create partitions
{
    oskit_error_t error;
    /* added min and max to create partitions */
    error = oskit_uvm_create(size, min, max,
                             &outproc->sp_vm);//DWC 4/5/2007
    if (error) {
      return error;
    }

    queue_init(&outproc->sp_thread_head);
    pthread_mutex_init(&outproc->sp_lock, NULL);

    /* Install our uvm handler */
    oskit_uvm_handler_set(outproc->sp_vm,
                          oskit_sproc_uvm_handler);

    outproc->sp_desc = desc;
    return 0;
}
```

F. UVM/UVM/OSKIT_UVM.C

The ‘uvm/uvm/oskit_uvm.c’ file was modified with the only change was to the ‘oskit_uvm_create’ function to permit partitioning of memory.

```
/*
 * API: Create a vmspace
 */
/* added changes to permit partitioning */
extern oskit_error_t
oskit_uvm_create(oskit_size_t size, oskit_size_t min,
                 oskit_size_t max,
                 oskit_vmspace_t *out_vm) //DWC 4/5/2007
{
    oskit_vmspace_t p;
    oskit_error_t error;

    if (min + size > max) { //DWC 4/5/2007
      return OSKIT_EINVAL;
    }

    error = oskit_uvm_vmspace_alloc(&p);
    if (error) {
      return error;
    }

    UVM_LOCK;
    /* allocate uvmspace by min and max of partition */
    p->vm_proc.p_vmspace = uvmspace_alloc(min,
                                min + size, 1); //DWC 4/5/2007
    UVM_UNLOCK;
    *out_vm = p;

    return 0;
}
```

G. EXAMPLES/X86/SPROC/KERNEL.H

The ‘examples/x86/sproc/kernel.h’ file contains the function prototypes and defines used for the ‘kernel.c’ file.

```
#ifndef _KERNEL_H_
#define _KERNEL_H_

/* # of iteration */
#define NITER         1
/* # of processes run in parallel */
#define NPROCESS 3
/* # of threads in a single process */
#define NTHREAD       1
/* user stack size */
#define USER_STACK_SIZE    (16*1024)
/* for printing of debugging content */
```

```
extern int oskit_sproc_debug;
/* initialization for system calls */
extern int syscall_init(void);
/* create process and run in user-domain */
static void     execute_process(void *);
/* trap handler to handle page faults */
static int handler(struct oskit_sproc_thread *sth,
                   int signo, int code,
                   struct trap_state *frame);

/* structure used for system calls */
extern struct oskit_sproc_sysent my_syscall_tab[];

#endif /*KERNEL.H*/
```

H. EXAMPLES/X86/SPROC/CONFIG.H

The ‘examples/x86/sproc/config.h’ file contains the structures and defines used by ‘kernel.c’ and ‘configapp.c’ for the configuration file structure.

```
#ifndef _CONFIG_H_
#define _CONFIG_H_


#define MAX_PROCESSES 20 //define for max # of processes
#define MAX_PARTITIONS 8 //define for max # of partitions
#define MAX_RESOURCES 20 //define for max # of resources
#define MAX_CVT 3 //not implemented yet
#define MAX_AUTHS 20 //not implemented yet
#define MAX_NAME_LEN 100 //define for policy names and
                         //process file names
#define DELAY 10 //define used for delaying program control

typedef enum {FALSE=0, TRUE} BOOL; //add Boolean type
/* access modes used for determining resource and partition
   flow privileges */
typedef enum {N=0, R=2, W=3, RW=5} E_ACCESS_MODE;
/* E_ABILITIES are not implemented yet */
typedef enum {RESTART=0, HALT, CONFIG_UPDATE} E_ABILITY;
/* structure contains internal information for processes */
struct oskit_sproc_info{
  char arg[MAX_NAME_LEN]; // string of c file of process
  int proc_size; //partition size
  long proc_min_addr; //partition min address for process
  long proc_max_addr; //partition max address for process
  long proc_heap_start_addr;
  long proc_heap_size;
  int sched_ticks; // ticks for scheduler
  pthread_attr_t *attr; //loaded during process create
  struct oskit_vmspace *sp_vm; //created during proc init
  int process_id; //pid of the process
};
/* structure containing information for partitions */
struct partition{
  int partition_id;
```

```
  int resource_ids[MAX_RESOURCES];
  int process_ids[MAX_PROCESSES];
  long part_min_addr;
  long part_size;
  int timeSlice;
};
/* structure containing information for processes */
struct process{
  int process_id;
  int priority;
  struct oskit_sproc_info info;
};
/* structure containing information for resources */
struct resource{
  int resource_id;
  int resource_type;
  long res_min_addr;
  long res_size;
};
/* structure not yet implemented */
struct sub_auth{
  int process_id; //aka "subject"
  E_ABILITY ability_id;
};
/* structure containing information for partition flows */
struct partFlowTuple{
  int partition_id_from;
  int partition_id_to;
  E_ACCESS_MODE access_mode;
};
/* structure containing information for resource flows */
struct resFlowTuple{
  int process_id;
  int resource_id;
  E_ACCESS_MODE access_mode;
};
/* struct of configuration file info for the prototype */
struct LPSKconfig{
  int id;  //aka "next"
  int version;
  int num_of_partitions;
  int num_of_processes;
  int num_of_resources;
  int num_of_part_flows;
  int num_of_res_flows;
  char policy[MAX_NAME_LEN];
  struct partition partitionList[MAX_PARTITIONS];
  struct resource resourceList[MAX_RESOURCES];
  struct sub_auth authList[MAX_AUTHS]; //not implemented
  struct process processList[MAX_PROCESSES];
  struct partFlowTuple
           partFlows[MAX_PARTITIONS*MAX_PARTITIONS];
  struct resFlowTuple
           resFlows[MAX_PROCESSES*MAX_RESOURCES];
};
/* structure not yet implemented */
struct LPSKConfigVectorTable{ //not implemented yet
```

```
  int version;
  int curr_conf_id;
  struct LPSKconfig CVT[MAX_CVT];
};

#endif /*CONFIG.H*/
```

I. EXAMPLES/X86/SPROC/USERMAIN_TESTSPROC.C

The ‘examples/x86/sproc/usermain_testsproc.c’ file contains the test cases used to cause the memory accesses that are trapped by the page fault trap handler.

```
/*
 * Copyright (c) 2001 The University of Utah and the Flux
 * Group.  All rights reserved.
 *
 * This file is part of the Flux OSKit.  The OSKit is free
 * software, also known as "open source;" you can
 * redistribute it and/or modify it under the terms of the
 * GNU General Public License (GPL), version 2, as
 * published by the Free Software Foundation (FSF).  To
 * explore alternate licensing terms, contact the
 * University of Utah at csl-dist@cs.utah.edu or +1-801-
 * 585-3271.
 * The OSKit is distributed in the hope that it will be
 * useful, but WITHOUT ANY WARRANTY; without even the
 * implied warranty of MERCHANTABILITY or FITNESS FOR A
 * PARTICULAR PURPOSE.  See the GPL for more details.  You
 * should have received a copy of the GPL along with the
 * OSKit; see the file COPYING.  If not, write to the FSF,
 * 59 Temple Place #330, Boston, MA 02111-1307, USA.
 */

/* Sample user program */

#include <oskit/c/stdio.h>
#include <oskit/c/malloc.h>
#include "user_syscall.h"

extern int errno;

int
main()
{
  int pid; //Process Id
  int rc;
  int value;
  char *val1, *val2;
  pid = syscall_getpid();
  printf("Start process (pid %d)\n", pid);

  syscall_sleep(6);
  value = 99;
  value++;
  /* Test Pushing Memory Access Class */
```

```
  asm volatile( "push 0xA0009FA8\n\t");
  /* Test Popping Memory Access Class */
  asm volatile( "pop 0xA0009FAC\n\t");
  asm volatile( "mov $0x12, %eax\n\t");
  /* Test Register to Memory Access Class */
  asm volatile( "mov %eax, 0xA0009FA8\n\t");
  /* Test Constant to Memory Access Class */
  *(int*)0xA0009FA8 = value;
  asm volatile( "push %eax\n\t");
  /* Test Pushing/Popping to Memory Address Access Class */
  asm volatile( "push $0xA000BEEF\n\t");
  asm volatile( "pop %eax\n\t");
  ////////////////////////////////////////////////////
  /* Test Memory Address to Register Access Class */
  asm volatile( "mov $0xA00DBEEF, %eax\n\t");
  asm volatile( "pop %eax\n\t");
  printf("value is: 0x%d", *(int*)0xA0009FA8);
  syscall_sleep(1);
  /* Test Memory to Register, Memory Pointed by a */
  /* Register to Memory, and Memory to Memory Pointed by */
  /* a Register Access Classes                    */
  memcpy(*(int*)0xA0009FAC, *(int*)0xA0009FA8, 4);
  /* Test Incrementing/Decrementing Memory Access Class */
  *(int*)0xA0009FA8 = *(int*)0xA0009FA8 + 1;
  *(int*)0xA0009FAC = *(int*)0xA0009FAC + 4;
  /* Test Pushing/Popping Memory Address Access Class */
  val1 = 0xA0009FA8;
  val2 = 0xA0009FAC;
  /* Test Memory to Memory Access Class */
  memmove(val1, val2, 1);

  syscall_sleep(2);
  /* Test Accessing Program Counter Address */
  asm volatile( "jmp 0xA0009FAC\n\t" );

#if 1
  /* Test userspace malloc */
  {
    int *base1, *base2;
    syscall_lock();
    base1 = (int*)malloc(10);
    syscall_unlock();
    printf("pid %d: malloc returns %p\n", pid, base1);
    syscall_lock();
    base2 = (int*)malloc(4096);
    syscall_unlock();
    printf("pid %d: malloc returns %p\n", pid, base2);
  }
#endif
  syscall_sleep(5);
  return 0;
}
```

J. EXAMPLES/X86/SPROC/ USERMAIN_HELLO.C

The ‘examples/x86/sproc/usermain_hello.c’ file contains the code to display the resources that are created and tested in the prototype.

```
/*
 * Copyright (c) 2001 The University of Utah and the Flux
 * Group.  All rights reserved.
 *
 * This file is part of the Flux OSKit.  The OSKit is free
 * software, also known as "open source;" you can
 * redistribute it and/or modify it under the terms of the
 * GNU General Public License (GPL), version 2, as
 * published by the Free Software Foundation (FSF).  To
 * explore alternate licensing terms, contact the
 * University of Utah at csl-dist@cs.utah.edu or +1-801-
 * 585-3271.
 * The OSKit is distributed in the hope that it will be
 * useful, but WITHOUT ANY WARRANTY; without even the
 * implied warranty of MERCHANTABILITY or FITNESS FOR A
 * PARTICULAR PURPOSE.  See the GPL for more details.  You
 * should have received a copy of the GPL along with the
 * OSKit; see the file COPYING.  If not, write to the FSF,
 * 59 Temple Place #330, Boston, MA 02111-1307, USA.
 */

/* Sample user program */

#include <oskit/c/stdio.h>
#include "user_syscall.h"

int main()
{
  int value2 = 899;
  int value = 455;
  int pid, tid;
  int i,j;
  /* obtain process id for displaying */
  pid = syscall_getpid();
  /* obtain thread id for displaying */
  tid = syscall_gettid();
  /* don’t run forever for testing */
  for (i = 0 ; i < 9; i++) {
    /* display process running results for testing */
    printf("proc 0x8%x, tid %2d: Hello World, iter = %d ",
           pid, tid, i);
    /* waste ticks without pre-emption */
    for(j = 0; j < 100000000; j++){}
    /* display resource values */
    printf("address of value is: %x value2 is: %x and value
         %d value2 %d \n", &value, &value2, value, value2);
    /* waste ticks with pre-emption */
    syscall_sleep(2);
  }
  printf("tid %d: Bye!\n", tid);
  /* tell kernel the process is done */
  return 0;
}
```

K. EXAMPLES/X86/SPROC/KERNEL.C

The ‘examples/x86/sproc/kernel.c’ file went through the most amount of changes. The changes made include reading the configuration file and creating partitions, resources, processes, resource flows, and partition flows. Other changes made involve adding a page fault trap handler that follows the rules of the flows to permit or deny access attempts made by processes.

```
/*
 * Copyright (c) 2001 The University of Utah and the Flux
 * Group.  All rights reserved.
 *
 * This file is part of the Flux OSKit.  The OSKit is free
 * software, also known as "open source;" you can
 * redistribute it and/or modify it under the terms of the
 * GNU General Public License (GPL), version 2, as
 * published by the Free Software Foundation (FSF).  To
 * explore alternate licensing terms, contact the
 * University of Utah at csl-dist@cs.utah.edu or +1-801-
 * 585-3271.
 * The OSKit is distributed in the hope that it will be
 * useful, but WITHOUT ANY WARRANTY; without even the
 * implied warranty of MERCHANTABILITY or FITNESS FOR A
 * PARTICULAR PURPOSE.  See the GPL for more details.  You
 * should have received a copy of the GPL along with the
 * OSKit; see the file COPYING.  If not, write to the FSF,
 * 59 Temple Place #330, Boston, MA 02111-1307, USA.
 */

/*
 * An example for simple process library.  This kernel
 * loads an ELF binary into a user space and execute it in
 * user mode.  Several quite simple system calls are
 * implemented.
 */
/* This file has been modified to add partitioning and
 * memory resources.  The code in this file makes up the
 * bulk of the LPSK prototype created from OSKit source
 */
#include <oskit/c/termios.h>
#include <oskit/c/unistd.h>
#include <oskit/clientos.h>
#include <oskit/debug.h>
#include <oskit/exec/exec.h>
#include <oskit/gdb.h>
#include <oskit/machine/pc/phys_lmm.h>
#include <oskit/sproc.h>
#include <oskit/startup.h>
#include <oskit/threads/pthread.h>
#include <oskit/version.h>
#include <oskit/x86/proc_reg.h>
#include <oskit/x86/trap.h>
#include <oskit/page.h>
```

```
#include <oskit/c/malloc.h>
#include <oskit/c/fcntl.h>
#include <oskit/io/absio.h>
#include <oskit/x86/pc/pic.h>
#include <oskit/x86/pc/pit.h>
#include <oskit/x86/pc/base_irq.h>
#include <uvm/sproc/sproc_internal.h>
#include <oskit/x86/eflags.h>
#include <oskit/lmm.h>
#include <stdio.h>

#include "proc.h"
#include "syscallno.h"
#include "kernel.h"
#include "config.h"
/* function to check access by comparing permitted flows
 * rules with requested access
 */
static int checkAccessFlows(oskit_size_t cr2,
                            E_ACCESS_MODE access_mode,
                            int sproc);
/* structure used to connect the page fault trap handler
 * in this file to be called when a page fault interrupt
 * occurs
 */
static struct oskit_sproc_desc process_desc = {
    NSYS,             /* # of system calls implemented */
    my_syscall_tab,         /* system call table */
    handler                 /* trap handler */
};
/* configuration file container*/
struct LPSKconfig *c_file;

/* main of LPSK prototype that reads the configuration file
 * and from the data gleaned create partitions, processes,
 * resource flows, resources, and partition flows
 */
extern int
main()
{
    int i, j;
    int fd;
    const char *path = "/configfile";
    oskit_off_t offset;
    oskit_size_t num_bytes;
    oskit_size_t * out;
    oskit_absio_t * absio;
    oskit_error_t error;
    pthread_attr_t threadattr;
    struct sched_param param;
#ifndef KNIT
     oskit_clientos_init_pthreads();
#endif
/* for GNU debugging and testing purposes */
#if 0
    {
     extern struct termios base_raw_termios,
```

```
                              base_cooked_termios;
      printf("setting serial port for gdb\n");
      base_raw_termios.c_ispeed = B38400;
      base_raw_termios.c_ospeed = B38400;
      base_cooked_termios.c_ispeed = B38400;
      base_cooked_termios.c_ospeed = B38400;
      gdb_pc_com_init(1, &base_raw_termios);

      gdb_trap_mask = (1 <<T_PAGE_FAULT) | (1 << T_NO_FPU);
      printf("break!\n");
      gdb_breakpoint();
      printf("go!\n");
    }
#endif
    /* start the clock */
    start_clock();
    /* instantiate environment for pthreads */
    start_pthreads();
    start_fs_bmod();

#if 0
    {
      extern int __isthreaded;
      __isthreaded = 1;          /* for freebsd libc */
    }
#endif
    /* start up virtual memory system plus swap file */
    printf(">> Initializing UVM\n");
    oskit_uvm_init();
    oskit_uvm_swap_init();
    printf(">> Swap On\n");
    if (swapon("/swapfile")) {
      extern int errno;
      panic("swapon failed (errno %d)\n", errno);
    }
    /* read the configuration file from the binary
     * created and linked into the elf image
     */
    printf(">> Starting to read config file\n");
    num_bytes = sizeof(struct LPSKconfig);

    fd = open(path, O_RDWR);
    if (fd == -1) {
      return -1;
    }
    printf(">> Opened file\n");
    error = fd_get_absio(fd, &absio);
    if (error) {
        printf("errror %i\n", error);
      return -1;
    }
    /* allocate space for configuration file container */
    c_file = (struct LPSKconfig *)
      malloc(sizeof(struct LPSKconfig));
    /* read configuration file */
    offset = 0;
    error = oskit_absio_read(absio, c_file,
```

```
                                offset, num_bytes, out);
    oskit_absio_release(absio); // don't need anymore
    if (error) {
      printf("errror %i\n", error);
      return -1;
    }
    printf("read config file!!!\n");
    getchar();
    /* start page daemon */
    printf(">> Starting the page daemon\n");
    oskit_uvm_start_pagedaemon();
    /* initialize simple process library */
    printf(">> Initializing Simple Process Library\n");
    oskit_sproc_init();

    printf(">> We are ready\n");

#ifdef  GPROF
    start_gprof();
#endif
    /* fill structure with pthread attributes */
    pthread_attr_init(&threadattr);

    /* Initialize my system calls */
    syscall_init();

    for (j = 0 ; j < NITER ; j++) {
      pthread_t th[c_file->num_of_processes];
      int rc;
      printf("******  Create processes (%d) ******\n", j);
      /* create processes */
      for (i = 0 ; i < c_file->num_of_processes ; i++) {
        /*
         * Create the thread.
         */
        /* set up details for scheduler for process */
        param.priority = PRIORITY_NORMAL;

        pthread_attr_setschedparam(&threadattr, &param);
        pthread_attr_setschedpolicy(&threadattr, SCHED_RR);

         threadattr.sched_ticks =
         c_file->processList[i].info.sched_ticks;
         c_file->processList[i].info.attr = &threadattr;
        /* add temp values for process id for ordering */
         c_file->processList[i].info.process_id = i;
        /* create thread for process */
         rc = pthread_create(&th[i], &threadattr,
             (void *(*)(void*))execute_process,
             (void *)&c_file->processList[i].info);
        assert(rc == 0);
      }
      /*
       * Delay for a while, while the threads prove
       * themselves.
       */
      oskit_pthread_sleep(DELAY);
```

```
      printf("********  Waiting (%d) ********\n", j);
      for (i = 0 ; i < c_file->num_of_processes ; i++) {
         pthread_join(th[i], NULL);
      }
    }
    printf("******** Terminated ********\n");
    return 0;
}
/* this function initializes the virtual memory mappings
 * for a process and starts processes
*/
static void
load_process(struct oskit_sproc *sproc,
             exec_info_t *exec_info, const char *elf,
             struct oskit_sproc_info *info)
{
    oskit_addr_t heap;
    oskit_error_t error;

    printf("**** creating a process [%s] (pid %p,
           thread %d) ****\n", elf, sproc,
           (int)pthread_self());
    /* create a process with the min and max boundaries of
     * the partition that the process resides
     */
    if (oskit_sproc_create(&process_desc, info->proc_size,
        info->proc_min_addr, info->proc_max_addr, sproc)) {
      panic("oskit_sproc_create failed\n");
    }
    error = oskit_sproc_load_elf(sproc, elf, exec_info);
    if (error) {
      panic("oskit_sproc_load_elf failed (0x%x)\n", error);
    }
    /* map heap area */
    //heap = HEAP_START_ADDR;
    heap = (oskit_size_t)info->proc_heap_start_addr;
    printf("heap location: %d", heap);
    /* create mapping for the process with the heap */
    error = oskit_uvm_mmap(sproc->sp_vm, &heap,
                   (oskit_size_t)info->proc_heap_size,
                   PROT_READ|PROT_WRITE,
                   MAP_PRIVATE|MAP_ANON|MAP_FIXED, 0, 0);
    if (error) {
      panic("oskit_uvm_mmap failed (heap)\n");
    }
    //assert(heap == HEAP_START_ADDR);
    assert(heap ==
           (oskit_size_t)info->proc_heap_start_addr);
    info->sp_vm = sproc->sp_vm;
}


/*********************************************************
 *
 *    Multiple threads in a Process
 *
 *********************************************************/
```

```
/* this structure is used to store process information */
struct arg {
    struct oskit_sproc     *sproc;
    oskit_addr_t entry;
    struct oskit_sproc_info *info;
};

/* this function initializes the user stack and page sizes
 * for a process and then puts the process in user mode
 * so that the process can begin
*/
static void
lwp(struct arg *arg)
{
    oskit_addr_t stkaddr;
    struct oskit_sproc_stack stk;
    /* pass arguments to process to know where heap
     * will be located for a process
     */
    oskit_size_t userarg[] = {
      (oskit_size_t)arg->info->proc_heap_start_addr,
      (oskit_size_t)arg->info->proc_heap_size, 300};
    stkaddr = 0;
    printf("sproc: %p user heap addr: %x \n", arg->sproc,
           (oskit_size_t)arg->info->proc_heap_start_addr);
    /* create the user process' stack */
    if (oskit_sproc_stack_alloc(arg->sproc, &stkaddr,
        USER_STACK_SIZE, PAGE_SIZE, &stk)) {
      panic("oskit_sproc_alloc_stack failed\n");
    }
    /* push the arguments for the process on it's stack */
    if (oskit_sproc_stack_push(&stk, &userarg,
        sizeof(userarg))) {
      panic("oskit_sproc_alloc_push failed\n");
    }

    printf("thread %d: process %p, user stack [%x..%x]\n",
           (int)pthread_self(), arg->sproc,
           stkaddr + PAGE_SIZE,
           stkaddr + USER_STACK_SIZE + PAGE_SIZE);
    /* switch to user-domain */
    oskit_sproc_switch(arg->sproc, arg->entry, &stk);
}

/* this function creates a process by initializing the
 * configuration file structure to have the new process
 * id's of the newly created process instead of
 * the placeholder id's from configuration file
 */
static void
execute_process(void *arg)
{
    struct oskit_sproc sproc;
    exec_info_t exec_info;
    int i, k, l;
    int process_id;
    const char *filename =
```

```
    (const char*)(((struct oskit_sproc_info *)(arg))->arg);
    pthread_t th[NTHREAD];

    printf("filename: %s \n", filename);
    load_process(&sproc, &exec_info, filename, arg);
    /* obtain the process id for the new process */
    process_id = (int)&sproc;
    printf("main proc 0x8%x!!!!!!!!\n", process_id);
    /* overwrite old process id placeholders in the
     * configuration file structure
     */
    i = ((struct oskit_sproc_info *)arg)->process_id;
    for(k = 0; k< c_file->num_of_partitions; k++){
      for(l = 0; l< c_file->num_of_processes; l++){
        if(c_file->processList[i].process_id ==
           c_file->partitionList[k].process_ids[l]){
             c_file->partitionList[k].process_ids[l] =
             process_id;
        }
      }
    }
    for(k = 0; k< c_file->num_of_res_flows; k++){
      if(c_file->processList[i].process_id ==
         c_file->resFlows[k].process_id){
        c_file->resFlows[k].process_id = process_id;
      }
    }
    c_file->processList[i].process_id = process_id;

    /* create the processes and let them go, killing them
     * when and if they return
     */
    for (i = 0 ; i < NTHREAD ; i++) {
      struct arg arg2;
      arg2.sproc = &sproc;
      arg2.entry = exec_info.entry;
      arg2.info = arg;

      pthread_create(&th[i],
                  ((struct oskit_sproc_info *)(arg))->attr,
                  (void*(*)(void*))lwp, &arg2);
    }
    for (i = 0 ; i < NTHREAD ; i++) {
      pthread_join(th[i], NULL);
    }

    printf("**** destroying process (thread %d) ****\n",
           (int)pthread_self());
    oskit_sproc_destroy(&sproc);
}

/* this function checks the resource flows and partition
 * flows from the configuration data to determine if the
 * given process can have the petitioned access
*/
static int checkAccessFlows(oskit_size_t cr2,
                            E_ACCESS_MODE access_mode,
```

```
                              int sproc){
int i, j, process_id;
int partition_from_id = 0;
int partition_to_id = 0;
int resource_id = 0;
int grant_access = 0;

process_id = sproc;
/* obtain partition where process resides */
printf("check access proc 0x8%x !!!!!!!!\n",process_id);
for(i=0;i<c_file->num_of_partitions;i++){
  for(j=0;j<c_file->num_of_processes;j++){
    if(c_file->partitionList[i].process_ids[j] ==
       process_id){
      partition_from_id =
      c_file->partitionList[i].partition_id;
      break;
    }
  }
}
/* obtain partition where access is attempted */
for(i=0;i<c_file->num_of_partitions;i++){
  if((cr2 >=
    (oskit_size_t)c_file->partitionList[i].part_min_addr)
    && (cr2 <=
     (oskit_size_t)c_file->partitionList[i].part_min_addr
    + (oskit_size_t)c_file->partitionList[i].part_size)){
    partition_to_id =
    c_file->partitionList[i].partition_id;
  }
}
//if no partition exists for read/write and from where,
//terminate
//the check on where, which is a sanity check
if((!partition_from_id) || (!partition_to_id)) return -1;

//First Check to see if there is a partition Flow to
//allow the access
for(i=0;i<c_file->num_of_part_flows;i++){
  if((c_file->partFlows[i].partition_id_from ==
      partition_from_id) &&
      (c_file->partFlows[i].partition_id_to ==
       partition_to_id) &&
     (((c_file->partFlows[i].access_mode) ==
     access_mode) ||
    ((c_file->partFlows[i].access_mode) >=
      access_mode+2))){
    grant_access = 1;
  }
}
if(!grant_access) return -1;

//Next Check to see if there is a resource Flow to allow
//the access
//to this process

for(i=0;i<c_file->num_of_resources;i++){
```

```
    if((cr2 >=
       (oskit_size_t)c_file->resourceList[i].res_min_addr)
       && (cr2 <=
       (oskit_size_t)c_file->resourceList[i].res_min_addr +
       (oskit_size_t)c_file->resourceList[i].res_size)){
      resource_id = c_file->resourceList[i].resource_id;
    }
  }

  //if no resource exists for read/write, terminate
  if(!resource_id) return -1;

  for(i=0;i<c_file->num_of_res_flows;i++){
    if((c_file->resFlows[i].process_id == process_id) &&
       (c_file->resFlows[i].resource_id == resource_id) &&
       (((c_file->resFlows[i].access_mode) == access_mode)
       || ((c_file->resFlows[i].access_mode) >=
           access_mode+2))){
      return 1; //Permission granted
    }
  }
  return -1; //Access denied
}

/* this function identifies a process that resides
 * in the attempted access location
*/
static int getVMSpace(oskit_size_t cr2){
  int i;
  for(i=0;i<c_file->num_of_processes;i++){
    printf("cr2: 0x%x >= min 0x%x cr2: 0x%x <=
           max 0x%x \n", cr2,
   (oskit_size_t)c_file->processList[i].info.proc_min_addr,
    cr2,
  (oskit_size_t)c_file->processList[i].info.proc_max_addr);

    if((cr2 >=
   (oskit_size_t)c_file->processList[i].info.proc_min_addr)
    && (cr2 <=
 (oskit_size_t)c_file->processList[i].info.proc_max_addr)){
      printf("going to swap vmspace\n");
      return i;
    }
  }
  return -1;
}

/*
 *  Trap handler for the simple process library
 *  This trap handler is implemented to be a page fault
 *  trap handler to catch the various memory access
 *  attempts
 */
static int
handler(struct oskit_sproc_thread *sthread, int signo,
int code, struct trap_state *ts)
{
```

```
oskit_size_t cr2 = get_cr2();
int i;
int from_user = (ts->cs & 3) || (ts->eflags & EFL_VM);
struct oskit_vmspace *processesVM;
long mem, val;
unsigned char data[ts->ecx];
E_ACCESS_MODE access_mode;

printf("handler proc 0x8%x !!!!!!!!\n",
       (int)sthread->st_process);
printf("eip address: %x \n", ts->eip);
printf("esp address: %x \n", ts->esp);
printf("cr2 address: %x \n", cr2);
/* Is this a program counter access attempt?
 * kill the process if it is
 */
if(ts->eip == cr2){
  printf("execute read or write so kill process!");
  return 1;//Don't allow execute! so kill process
}
/* Is this a nested trap or double fault?
 * kill the process as to not allow if it is
 */
if(!from_user){
  printf("kernel\n");
  return 1; //Don't allow trap nesting or double
            //faults! So kill process
}

printf("ERR!: thread = %d, process = %p, signo = %d, "
     "code = %d, frame = %p\n", (int)pthread_self(),
      sthread->st_process, signo, code, ts);
printf("page fault address 0x%x\n", cr2);
/* if page fault from accessing kernel memory
 * kill process because this should never happen
 */
/* sanity check */
if(cr2 < OSKIT_UVM_MINUSER_ADDRESS) return 1;
/* check if partition exists, if not kill process */
if((i = getVMSpace(cr2)) == -1) return 1;
/* get mapping from other process to use */
processesVM = c_file->processList[i].info.sp_vm;
printf("opcode: %x \n", *(unsigned short*)ts->eip);
switch(*(unsigned short*)(ts->eip)){
  case(0x05FF): //incl addr
      /* define access mode of this opcode */
      access_mode = W;
      /* grab memory address accessed */
      mem = *(long*)(ts->eip+2);
      printf("0x05FF!\n");
      /* make sure the address access is the fault
       * address
       */
      if(mem != cr2) return 1; //sanity check
      //check access and kill if denied
      if((i = checkAccessFlows(cr2, access_mode,
         (int)sthread->st_process)) == -1) return 1;
```

```
    //swap in memory mapping to access memory
    processesVM = oskit_uvm_vmspace_set(processesVM);
    /* perform operation of instruction attempted by
     * user since it has enough permissions
     */
    *(int*)mem = *(int*)mem + 1;
    /* swap in old mapping */
    oskit_uvm_vmspace_set(processesVM);
    /* set user to return after instruction since
     * we have already executed the instruction
     */
    ts->eip = ts->eip+6;
    return 0; //handled trap so return to process
case(0x0583): //addl const,addr
    /* define access mode of this opcode */
    access_mode = W;
    /* grab memory address accessed */
    mem = *(long*)(ts->eip+2);
    val = *(unsigned char*)(ts->eip+6);
    printf("0x0583!\n");
    if(mem != cr2) return 1; //sanity check
    //check access and kill if denied
    if((i = checkAccessFlows(cr2, access_mode,
       (int)sthread->st_process)) == -1) return 1;
    //swap in memory mapping to access memory
    processesVM = oskit_uvm_vmspace_set(processesVM);
    /* perform operation of instruction attempted by
     * user since it has enough permissions
     */
    *(int*)mem = *(int*)mem + val;
    /* swap in old mapping */
    oskit_uvm_vmspace_set(processesVM);
    /* set user to return after instruction since
     * we have already executed the instruction
     */
    ts->eip = ts->eip+7;
    return 0; //handled trap so return to process
case(0x35FF): //pushl addr
    /* define access mode of this opcode */
    access_mode = R;
    /* grab memory address accessed */
    mem = *(long*)(ts->eip+2);
    //val = *(unsigned char*)(ts->eip+6);
    printf("0x058F!\n");
    if(mem != cr2) return 1; //sanity check
    //check access and kill if denied
    if((i = checkAccessFlows(cr2, access_mode,
       (int)sthread->st_process)) == -1) return 1;
    /* increase user stack since
     * we need to execute the instruction
     */
    ts->esp = ts->esp - 4;
    //swap in memory mapping to access memory
    processesVM = oskit_uvm_vmspace_set(processesVM);
    /* perform operation of instruction attempted by
     * user since it has enough permissions
     */
```

```
    val = *(long*)mem;
    /* swap in old mapping */
    oskit_uvm_vmspace_set(processesVM);
    *(long*)(ts->esp) = val;
    /* set user to return after instruction since
     * we have already executed the instruction
     */
    ts->eip = ts->eip+6;
    return 0; //handled trap so return to process
case(0x058F): //popl addr
    /* define access mode of this opcode */
    access_mode = W;
    /* grab memory address accessed */
    mem = *(long*)(ts->eip+2);
    val = *(long*)(ts->esp);
    printf("0x058F!\n");
    if(mem != cr2) return 1; //sanity check
    //check access and kill if denied
    if((i = checkAccessFlows(cr2, access_mode,
       (int)sthread->st_process)) == -1) return 1;
    //swap in memory mapping to access memory
    processesVM = oskit_uvm_vmspace_set(processesVM);
    /* perform operation of instruction attempted by
     * user since it has enough permissions
     */
    *(long*)mem = val;
    /* swap in old mapping */
    oskit_uvm_vmspace_set(processesVM);
    /* decrease user stack since
     * we have already executed the instruction
     */
    ts->esp = ts->esp + 4;
    /* set user to return after instruction since
     * we have already executed the instruction
     */
    ts->eip = ts->eip+6;
    return 0; //handled trap so return to process
case(0x05c7): //movl const, addr
    /* define access mode of this opcode */
    access_mode = W;
    /* grab memory address accessed */
    mem = *(long*)(ts->eip+2);
    val = *(long*)(ts->eip+6);
    printf("0x05c7!\n");
    if(mem != cr2) return 1; //sanity check
    //check access and kill if denied
    if((i = checkAccessFlows(cr2, access_mode,
       (int)sthread->st_process)) == -1) return 1;
    //swap in memory mapping to access memory
    processesVM = oskit_uvm_vmspace_set(processesVM);
    /* perform operation of instruction attempted by
     * user since it has enough permissions
     */
    //*(int*)mem = val;
    asm volatile(
            "pushl %%eax\n\t"
            "pushl %%ebx\n\t"
```

```
            "pushl %1\n\t"
            "pushl %0\n\t"
            "popl %%eax\n\t"
            "popl %%ebx\n\t"
            "movl %%eax, (%%ebx)\n\t"
            "popl %%ebx\n\t"
            "popl %%eax\n\t"
            : "=m" ((long)val)
            : "m" ((oskit_size_t)mem)
            );
    /* swap in old mapping */
    oskit_uvm_vmspace_set(processesVM);
    /* set user to return after instruction since
     * we have already executed the instruction
     */
    ts->eip = ts->eip+10;
    return 0; //handled trap so return to process
case(0x008b): //mov (%eax),%eax
    /* define access mode of this opcode */
    access_mode = R;
    /* grab memory address accessed */
    mem = ts->eax;
    printf("0x008b!\n");
    if(mem != cr2) return 1; //sanity check
    //check access and kill if denied
    if((i = checkAccessFlows(cr2, access_mode,
       (int)sthread->st_process)) == -1) return 1;
    //swap in memory mapping to access memory
    processesVM = oskit_uvm_vmspace_set(processesVM);
    /* perform operation of instruction attempted by
     * user since it has enough permissions
     */
    ts->eax = *(int*)mem;
    /* swap in old mapping */
    oskit_uvm_vmspace_set(processesVM);
    /* set user to return after instruction since
     * we have already executed the instruction
     */
    ts->eip = ts->eip+2;
    return 0; //handled trap so return to process
case(0x0289): //mov %eax,(%edx)
    /* define access mode of this opcode */
    access_mode = W;
    /* grab memory address accessed */
    mem = ts->edx;
    val = ts->eax;
    printf("0x0289!\n");
    if(mem != cr2) return 1; //sanity check
    //check access and kill if denied
    if((i = checkAccessFlows(cr2, access_mode,
       (int)sthread->st_process)) == -1) return 1;
    //swap in memory mapping to access memory
    processesVM = oskit_uvm_vmspace_set(processesVM);
    /* perform operation of instruction attempted by
     * user since it has enough permissions
     */
    *(int*)mem = val;
```

```
    /* swap in old mapping */
    oskit_uvm_vmspace_set(processesVM);
    /* set user to return after instruction since
     * we have already executed the instruction
     */
    ts->eip = ts->eip+2;
    return 0; //handled trap so return to process
case(0x158b): //mov addr, %edx
    /* define access mode of this opcode */
    access_mode = N;
    /* grab memory address accessed */
    mem = *(long*)(ts->eip+2);
    printf("0x158b!\n");
    if(mem != cr2) return 1; //sanity check
    //check access and kill if denied
    if((i = checkAccessFlows(cr2, access_mode,
       (int)sthread->st_process)) == -1) return 1;
    //swap in memory mapping to access memory
    processesVM = oskit_uvm_vmspace_set(processesVM);
    /* perform operation of instruction attempted by
     * user since it has enough permissions
     */
    ts->edx = mem;
    /* swap in old mapping */
    oskit_uvm_vmspace_set(processesVM);
    /* set user to return after instruction since
     * we have already executed the instruction
     */
    ts->eip = ts->eip+6;
    return 0; //handled trap so return to process
case(0xa4f3): //repz movsb %ds:(%esi), %es:(%edi)
    //based on ecx count
    /* define access mode of the first half opcode */
    access_mode = R;
    printf("esi address: %x \n", ts->esi);
    printf("edi address: %x \n", ts->edi);
    /* grab memory address accessed */
    mem = ts->esi;
    printf("0xa4f3!\n");
    if(mem != cr2) return 1; //sanity check
    //check access and kill if denied
    if((i = checkAccessFlows(cr2, access_mode,
       (int)sthread->st_process)) == -1) return 1;
    //swap in memory mapping to access memory
    processesVM = oskit_uvm_vmspace_set(processesVM);
    /* perform operation of instruction attempted by
     * user since it has enough permissions
     */
    for(i=0;i<(int)ts->ecx;i++)
      data[i] = (unsigned char)*(((long*)ts->esi)+i);
    /* swap in old mapping */
    oskit_uvm_vmspace_set(processesVM);
    //from and to addresses may reside in separate
    //partitions
    //so we must set vmspace again
    //if no partition found, kill process
    if((i = getVMSpace(ts->edi)) == -1) return 1;
```

```
    //swap in memory mapping to access memory
    processesVM = c_file->processList[i].info.sp_vm;
    /* define access mode of this half of opcode */
    access_mode = W;
    //check access and kill if denied
    if((i = checkAccessFlows(cr2, access_mode,
       (int)sthread->st_process)) == -1) return 1;
    //swap in memory mapping to access memory
    processesVM = oskit_uvm_vmspace_set(processesVM);
    /* perform operation of instruction attempted by
     * user since it has enough permissions
     */
    for(i=0;i<(int)ts->ecx;i++)
      *(((long*)ts->edi)+i) = data[i];
    /* swap in old mapping */
    oskit_uvm_vmspace_set(processesVM);
    /* set user to return after instruction since
     * we have already executed the instruction
     */
    ts->eip = ts->eip+2;
    return 0; //handled trap so return to process
default:
switch(*(unsigned char*)(ts->eip)){
    case(0x58): //pop addr value into eax register
      //this will not occur
      /* define access mode of this opcode */
      access_mode = N;
      printf("0x58!\n");
      //check access and kill if denied
      if((i = checkAccessFlows(cr2, access_mode,
         (int)sthread->st_process)) == -1) return 1;
      //swap in memory mapping to access memory
      processesVM =
      oskit_uvm_vmspace_set(processesVM);
      /* perform operation of instruction attempted
      * by user since it has enough permissions
      */
      ts->eax = cr2;
      /* swap in old mapping */
      oskit_uvm_vmspace_set(processesVM);
      /* set user to return after instruction since
      * we have already executed the instruction
      */
      ts->eip = ts->eip+1;
      return 0; //handled trap so return to process
    case(0x68): //push addr code will not cause trap
      //so this will never happen
      /* define access mode of this opcode */
      access_mode = N;
      /* grab memory address accessed */
      mem = *(long*)(ts->eip+1);
      val = 0;//ts->eax;
      printf("0x68!\n");
      if(mem != cr2) return 1; //sanity check
      //check access and kill if denied
      if((i = checkAccessFlows(cr2, access_mode,
         (int)sthread->st_process)) == -1) return 1;
```

```
  //swap in memory mapping to access memory
  processesVM =
  oskit_uvm_vmspace_set(processesVM);
  /* perform operation of instruction attempted
  * by user since it has enough permissions
  */
  //ts->eax = mem;
  asm volatile(
       "pushl %1\n\t"
       : "=m" (val)
       : "m" (mem)
       );
  /* swap in old mapping */
  oskit_uvm_vmspace_set(processesVM);
  /* set user to return after instruction since
  * we have already executed the instruction
  */
  ts->eip = ts->eip+5;
  return 0; //handled trap so return to process
case(0xa1): //mov addr, %eax
  /* define access mode of this opcode */
  access_mode = N;
  /* grab memory address accessed */
  mem = *(long*)(ts->eip+1);
  val = 0;//ts->eax;
  printf("0xa1!\n");
  if(mem != cr2) return 1; //sanity check
  //check access and kill if denied
  if((i = checkAccessFlows(cr2, access_mode,
     (int)sthread->st_process)) == -1) return 1;
  //swap in memory mapping to access memory
  processesVM =
  /* swap in old mapping */
  oskit_uvm_vmspace_set(processesVM);
  /* perform operation of instruction attempted
  * by user since it has enough permissions
  */
  //ts->eax = mem;
  asm volatile(
       "pushl %%ebx\n\t"
       "pushl %1\n\t"
       "popl %%ebx\n\t"
       "mov %%ebx, %0\n\t"
       "popl %%ebx\n\t"
       : "=m" (val)
       : "m" (mem)
       );
  oskit_uvm_vmspace_set(processesVM);
  ts->eax = val;
  /* set user to return after instruction since
  * we have already executed the instruction
  */
  ts->eip = ts->eip+5;
  return 0; //handled trap so return to process
case(0xa3): //mov %eax, addr
  /* define access mode of this opcode */
  access_mode = R;
```

```
                /* grab memory address accessed */
                mem = *(long*)(ts->eip+1);
                val = ts->eax;
                printf("0xa3!\n");
                if(mem != cr2) return 1; //sanity check
                //check access and kill if denied
                if((i = checkAccessFlows(cr2, access_mode,
                   (int)sthread->st_process)) == -1) return 1;
                //swap in memory mapping to access memory
                processesVM =
                oskit_uvm_vmspace_set(processesVM);
                /* perform operation of instruction attempted
                * by user since it has enough permissions
                */
                //*(int*)mem = ts->eax;
                asm volatile(
                     "pushl %%eax\n\t"
                     "pushl %%ebx\n\t"
                     "pushl %1\n\t"
                     "pushl %0\n\t"
                     "popl %%ebx\n\t"
                     "popl %%eax\n\t"
                     "mov %%ebx, (%%eax)\n\t"
                     "popl %%ebx\n\t"
                     "popl %%eax\n\t"
                     : "=m" (val)
                     : "m" ((oskit_size_t)mem)
                     );
                /* swap in old mapping */
                oskit_uvm_vmspace_set(processesVM);
                /* set user to return after instruction since
                * we have already executed the instruction
                */
                ts->eip = ts->eip+5;
                return 0; //handled trap so return to process
              default: printf("not found!\n");
                return 1; //not found so kill process
            }
        }
        return 0; //handled trap so return to process
    }

    /*
     *  For debugging
     */
    void
    dump_kvmspace()
    {
        oskit_uvm_vmspace_dump(&oskit_uvm_kvmspace);
    }
```

L. EXAMPLES/X86/SPROC/USER_CRT.C

The ‘examples/x86/sproc/user_crt.c’ file was modified to permit heap allocation within partitions that the user process resides.

```
/*
 * Copyright (c) 2001 The University of Utah and the Flux
 * Group.  All rights reserved.
 *
 * This file is part of the Flux OSKit.  The OSKit is free
 * software, also known as "open source;" you can
 * redistribute it and/or modify it under the terms of the
 * GNU General Public License (GPL), version 2, as
 * published by the Free Software Foundation (FSF).  To
 * explore alternate licensing terms, contact the
 * University of Utah at csl-dist@cs.utah.edu or +1-801-
 * 585-3271.
 * The OSKit is distributed in the hope that it will be
 * useful, but WITHOUT ANY WARRANTY; without even the
 * implied warranty of MERCHANTABILITY or FITNESS FOR A
 * PARTICULAR PURPOSE.  See the GPL for more details.  You
 * should have received a copy of the GPL along with the
 * OSKit; see the file COPYING.  If not, write to the FSF,
 * 59 Temple Place #330, Boston, MA 02111-1307, USA.
 */

/*
 * C run time for user mode process.
 * Initialize minimal C library, allocate malloc arena,
 * etc.
 */

#include <oskit/c/stdio.h>
#include <oskit/com/mem.h>
#include <oskit/lmm.h>
#include <oskit/c/malloc.h>
#include <oskit/c/environment.h> /* libc_memory_object */
#include <oskit/c/unistd.h>/* exit */
#include <oskit/uvm.h>

#include "proc.h"
#include "user_syscall.h"

lmm_t malloc_lmm = LMM_INITIALIZER;
struct lmm_region region;

extern void syscall_return(int code);
extern int main(int argc, char **argv);
/* pass paramaters to user processes for heap allocation
 * within their partition
 */
extern int
_start(int arg1, int arg2, int arg3)
{
    oskit_mem_t *memi;
    static volatile int initstate = 0;

    /* Poor lock */
    syscall_lock();
    if (initstate == 0) {

      memi = oskit_mem_init();
```

```
        printf("Start user process\n");
        /* Print the arguments received from the kernel */
        printf("arg1 = %x, arg2 = %x, arg3 = %x\n", arg1,
               arg2, arg3);

        /* Initialize LMM for userspace malloc heap */
         /* added heap allocation based on partition space */
        lmm_add_region(&malloc_lmm, &region, (void*)arg1,
                       arg2, 0, 0); //DWC 4/6/2007
        lmm_add_free(&malloc_lmm, (void*)arg1,
                     arg2); //DWC 4/6/2007

        libc_memory_object = memi;

        /* set exit hook */
        oskit_libc_exit = syscall_return;
        initstate = 1;
      }
      syscall_unlock();

      /* XXX: please someone add argc and argv! */
      exit(main(0, NULL));
}
```

M. EXAMPLES/X86/SPROC/CONFIGAPP.C

The ‘examples/x86/sproc/configapp.c’ file contains the code of the configapp application that creates configuration files in binary form.

```
#include <stdio.h>
#include <fcntl.h>
#include "config.h"

/* configuration file structure used by LPSK prototype */
static struct LPSKconfig *c_file;

int main(){
  /* allocate space for the config file structure */
  c_file = (struct LPSKconfig *) malloc(sizeof(struct
           LPSKconfig));
  /* set unique configuration file id */
  c_file->id = 100;
  /* set policy name */
  strncpy(c_file->policy,"policy1", 7);
  /* set version number */
  c_file->version = 1;
  /* set number of partitions */
  c_file->num_of_partitions = 2;
  /* set number of resources */
  c_file->num_of_resources = 2;
  /* set number of processes */
  c_file->num_of_processes = 2;
  /* set number of partition flows */
  c_file->num_of_part_flows = 1;
```

```
/* set number of resource flows */
c_file->num_of_res_flows = 2;
/* set timeSlices used for scheduling for partitions */
c_file->partitionList[0].timeSlice = 1;
c_file->partitionList[1].timeSlice = 1;
/* set process id's for the processes */
c_file->processList[0].process_id = 153;
c_file->processList[1].process_id = 253;
/* set priorities used for scheduling the processes */
c_file->processList[0].priority = 1;
c_file->processList[1].priority = 1;
/* set partition memory locations and their sizes*/
c_file->partitionList[0].part_min_addr = 0x90000000;
c_file->partitionList[0].part_size = 0x5000000;
c_file->partitionList[1].part_min_addr = 0xA0000000;
c_file->partitionList[1].part_size = 0x10000000;
/* set unique resource id's for each resource */
c_file->resourceList[0].resource_id = 12345;
c_file->resourceList[1].resource_id = 6785;
/* set resource memory locations and their sizes*/
c_file->resourceList[0].res_min_addr = 0xA0009FA8;
c_file->resourceList[0].res_size = 0x4;
c_file->resourceList[1].res_min_addr = 0xA0009FAC;
c_file->resourceList[1].res_size = 0x4;
/* pass unique resource id's to partitions */
c_file->partitionList[1].resource_ids[0] = 12345;
c_file->partitionList[1].resource_ids[1] = 6785;
/* pass unique process id's to partitions */
c_file->partitionList[0].process_ids[0] = 153;
c_file->partitionList[1].process_ids[0] = 253;
/* set unique partition id's for each partition */
c_file->partitionList[0].partition_id = 89354;
c_file->partitionList[1].partition_id = 90876;
/* pass unique partition id's to partition flows */
c_file->partFlows[0].partition_id_from = 89354;
c_file->partFlows[0].partition_id_to = 90876;
/* set the access mode for partition flow */
c_file->partFlows[0].access_mode = RW;
/* pass unique resource id's to resource flows */
c_file->resFlows[0].resource_id = 12345;
c_file->resFlows[1].resource_id = 6785;
/* define the process that has access to the resource
 * flows
 */
c_file->resFlows[0].process_id = 153;
c_file->resFlows[1].process_id = 153;
/* set the access modes for the resource flows */
c_file->resFlows[0].access_mode = RW;
c_file->resFlows[1].access_mode = RW;
/* set the name of the file with main of the process */
strncpy(c_file->processList[0].info.arg,
       "/usermain_testsproc", 19);
/* set partition constraints to the process mapping
 * defines and scheduling ticks of the process
 */
c_file->processList[0].info.proc_size = 0x5000000;
c_file->processList[0].info.proc_min_addr = 0x90000000;
```

```
  c_file->processList[0].info.proc_max_addr = 0x95000000;
  c_file->processList[0].info.proc_heap_start_addr =
    0x92500000;
  c_file->processList[0].info.proc_heap_size = 0x10000;
  c_file->processList[0].info.sched_ticks = 1;
  /* set the name of the file with main of the process */
  strncpy(c_file->processList[1].info.arg,
        "/usermain_hello", 15);
  /* set partition constraints to the process mapping
   * defines and scheduling ticks of the process
   */
  c_file->processList[1].info.proc_size = 0x10000000;
  c_file->processList[1].info.proc_min_addr = 0xA0000000;
  c_file->processList[1].info.proc_max_addr = 0xB0000000;
  c_file->processList[1].info.proc_heap_start_addr =
    0xA7500000;
  c_file->processList[1].info.proc_heap_size = 0x10000;
  c_file->processList[1].info.sched_ticks = 1;
  /* write the config data to the configuration file */
  writeData(c_file, "configfile");
  return 0;
}

/* Writing */
/* This function writes the LPSKconfig structure to file */
int writeData( struct LPSKconfig* data, const char * file)
{
   int fd = 0;
   FILE *f;
   /* open the file for writing */
   f = fopen(file, "w");
   /* write the binary structure right to the file */
   fwrite(data,sizeof(struct LPSKconfig),1,f);
   /* close the file */
   fclose(f);

   return 0;
}
```

N. EXAMPLES/X86/SPROC/GNUMAKERULES

The ‘examples/x86/sproc/gnumakerules’ was modified to allow processes to run in their specific partition and to read the configuration file for the LPSK prototype.

```
/*
 * Copyright (c) 2001 The University of Utah and the Flux
 * Group.  All rights reserved.
 *
 * This file is part of the Flux OSKit.  The OSKit is free
 * software, also known as "open source;" you can
 * redistribute it and/or modify it under the terms of the
 * GNU General Public License (GPL), version 2, as
 * published by the Free Software Foundation (FSF).  To
 * explore alternate licensing terms, contact the
```

```
 * University of Utah at csl-dist@cs.utah.edu or +1-801-
 * 585-3271.
 * The OSKit is distributed in the hope that it will be
 * useful, but WITHOUT ANY WARRANTY; without even the
 * implied warranty of MERCHANTABILITY or FITNESS FOR A
 * PARTICULAR PURPOSE.  See the GPL for more details.  You
 * should have received a copy of the GPL along with the
 * OSKit; see the file COPYING.  If not, write to the FSF,
 * 59 Temple Place #330, Boston, MA 02111-1307, USA.
 */

ifndef _oskit_examples_x86_sproc_makerules_
_oskit_examples_x86_sproc_makerules__ = yes

USER_PROGS = usermain_testsproc usermain_hello
             usermain_malloc
# added configfile as a bmod to link in configuration file
BMODS = kernel swapfile configfile $(USER_PROGS)
TARGETS = Image

all: $(TARGETS)

SRCDIRS += $(OSKIT_SRCDIR)/examples/x86/sproc

INCDIRS += $(OSKIT_SRCDIR)/oskit/c

CLEAN_FILES +=  $(TARGETS) $(BMODS) *.gdb
OSKIT_CFLAGS += -DOSKIT

#
# The C library is made up of several pieces. The core
# library, the POSIX interface that converts syscalls to
# COM, and the actual COM interfaces. Note that the C
# library is built with the COM library.
CLIB   = -loskit_c
#CLIB_P = -loskit_c_p -loskit_gprof -loskit_c_p -
#loskit_kern_p

include $(OSKIT_SRCDIR)/GNUmakerules

DEPENDLIBS = $(filter %.a,
       $(foreach DIR,$(LIBDIRS),$(wildcard $(DIR)/*)))

THRDLIBS   = -loskit_threads -loskit_netbsd_uvm
THRDLIBS_P   = -loskit_threads_p -loskit_netbsd_uvm_p
CLIB   = -loskit_freebsd_c_r -loskit_com -loskit_threads
CLIB_P = -loskit_freebsd_c_r_p -loskit_com_p \
         -loskit_threads_p -loskit_gprof \
          -loskit_freebsd_c_r_p -loskit_kern_p

kernel: $(OBJDIR)/lib/multiboot.o kernel.o kern_syscall.o
     $(DEPENDLIBS)
      $(OSKIT_QUIET_MAKE_INFORM) "Linking example $@"
      $(LD) -Ttext 100000 $(LDFLAGS) $(OSKIT_LDFLAGS) \
            -o $@ $(filter-out %.a,$^)       \
            -loskit_startup -loskit_fsnamespace \
            -loskit_memfs -loskit_sproc \
```

```
          -loskit_netbsd_uvm -loskit_exec -loskit_memfs \
          $(THRDLIBS) -loskit_clientos \
          -loskit_dev -loskit_kern -loskit_lmm \
          $(CLIB) $(OBJDIR)/lib/crtn.o

kernel_p.gdb: $(OBJDIR)/lib/multiboot.o kernel.po
      kern_syscall.po $(DEPENDLIBS)
     $(OSKIT_QUIET_MAKE_INFORM) "Linking example $@"
     $(LD) -Ttext 100000 $(LDFLAGS) $(OSKIT_LDFLAGS) \
          -o $@ $(filter-out %.a,$^)       \
          -loskit_startup_p -loskit_fsnamespace_p \
          -loskit_memfs_p -loskit_sproc_p \
          -loskit_netbsd_uvm_p -loskit_exec_p \
          -loskit_memfs_p \
          $(THRDLIBS_P) -loskit_clientos_p \
          -loskit_realtime_p -loskit_kern_p -loskit_lmm_p\
          $(CLIB_P) \
          $(OBJDIR)/lib/crtn.o

kernel_p: kernel_p.gdb
     cp kernel_p.gdb kernel_p
     strip kernel_p

#
# Build user mode programs
#
USER_CRT  = user_crt.o
USER_OBJS = user_syscall.o user_mem.o user_morecore.o
# added 90000000 to have this process be in a partition
usermain_testsproc: $(USER_CRT) usermain_testsproc.o
  $(USER_OBJS) \
  $(CC) -nostdlib -static \
  $(USER_CRT) usermain_testsproc.o \
  $(USER_OBJS) \
          -o $@ -Xlinker -Ttext -Xlinker 90000000 \
          -L../../../lib -loskit_c -loskit_lmm
# added A0000000 to have this process be in a partition
usermain_hello: $(USER_CRT) usermain_hello.o $(USER_OBJS) \
     $(CC) -nostdlib -static \
      $(USER_CRT) usermain_hello.o $(USER_OBJS) \
          -o $@ -Xlinker -Ttext -Xlinker A0000000 \
          -L../../../lib -loskit_c -loskit_lmm
# added 50000000 to have this process be in a partition
usermain_malloc: $(USER_CRT) usermain_malloc.o $(USER_OBJS)
     $(CC) -nostdlib -static $(USER_CRT) usermain_malloc.o
     $(USER_OBJS) \
          -o $@ -Xlinker -Ttext -Xlinker 50000000 \
          -L../../../lib -loskit_c -loskit_lmm

#
# Small swapfile (1MB)
#
swapfile:
     dd if=/dev/zero of=$@ count=2048

#
# Bind all together!
```

```
#
Image: $(BMODS)
      echo "use 'mkmb2 $(BMODS)' to build the bmod"

Image_p: kernel_p kernel_p.gdb swapfile $(USER_PROGS)
      echo "use 'mkmb2 -o $@ kernel_p kernel_p.gdb:a.out
      swapfile configfile $(USER_PROGS)' to build a bmod"

endif
```

O. DISSASSEMBLY OF TESTSPROC.C

The ‘dump’ file of testpsroc.c was created using GNU objdump. The machine code and the assembly are listed with comments to understand which assembly tested which memory access class test. The code has comments of which test was run by the assembly instruction and follow the naming convention used in Appendix B. The opcodes mentioned in Appendix B are listed here as machine code and are either the first byte or first two bytes listed on each line.

```
4c:   ff 35 a8 9f 00 a0     pushl 0xa0009fa8     //Test AC10
52:   8f 05 ac 9f 00 a0     popl 0xa0009fac       //Test AC11
58:   b8 12 00 00 00        mov $0x12,%eax
5d:   a3 a8 9f 00 a0        mov %eax,0xa0009fa8  //Test AC9
62:   c7 05 a8 9f 00 a0 64  movl $0x64,0xa0009fa8//Test AC1
69:   00 00 00
6c:   50                    push %eax
6d:   68 ef be 00 a0        push $0xa000beef     //Test AC6
72:   58                    pop %eax//Test AC6
73:   b8 ef be 0d a0        mov $0xa00dbeef,%eax //Test AC2
78:   58                    pop %eax
79:   58                    pop %eax
7a:   5a                    pop %edx
7b:   6a 64                 push $0x64
7d:   68 4f 00 00 00        push $0x4f
82:   e8 fc ff ff ff        call 83 <main+0x83>
87:   c7 04 24 01 00 00 00  movl $0x1,(%esp,1)
8e:   e8 fc ff ff ff        call 8f <main+0x8f>
93:   a1 a8 9f 00 a0        mov 0xa0009fa8,%eax   //Test AC8
98:   8b 15 ac 9f 00 a0     mov 0xa0009fac,%edx   //Test AC8
9e:   8b 00                 mov (%eax),%eax       //Test AC3
a0:   89 02                 mov %eax,(%edx)       //Test AC4
a2:   83 c4 0c              add $0xc,%esp
a5:   ff 05 a8 9f 00 a0     incl 0xa0009fa8       //Test AC5
ab:   83 05 ac 9f 00 a0 04  addl $0x4,0xa0009fac  //Test AC5
b2:   6a 01                 push $0x1
b4:   68 ac 9f 00 a0        push $0xa0009fac      //Test AC7
b9:   68 a8 9f 00 a0        push $0xa0009fa8      //Test AC7
be:   e8 fc ff ff ff        call bf <main+0xbf>   //Test AC7
c3:   c7 04 24 02 00 00 00  movl $0x2,(%esp,1)    //Test AC7
ca:   e8 fc ff ff ff        call cb <main+0xcb>   //Test AC7
/* Test AC12 */
```

```
cf:   e9 a8 9f 00 a0        jmp a000a07c <main+0xa000a07c>
```

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

1. Saltzer, J. H., and Schroeder, M. D., “The Protection of Information in Operating Systems,” *Proceedings of the IEEE*. 63(9):1278-1308. 1975.

2. Nguyen, T. D., Levin, T. E., and Irvine, C. E., "TCX Project: High Assurance for Secure Embedded Systems," *11th IEEE Real-Time and Embedded Technology and Applications Symposium Work-In-Progress Session*, San Francisco, CA, March 2005.

3. Levin, T. E., Irvine, C. E., and Nguyen, T. D., "A Least Privilege Model for Static Separation Kernels," NPS-CS-05-003, Naval Postgraduate School, October 2004.

4. Irvine, C. E., Levin, T. E., Nguyen, T. D., and Dinolt, G. W., "The Trusted Computing Exemplar Project," Proceedings of the 2004 IEEE Systems, Man and Cybernetics Information Assurance Workshop, West Point, NY, June 2004, pp. 109-115.

5. Myers, P. A., “Subversion: The Neglected Aspect of Computer Security,” Master’s Thesis, Naval Postgraduate School, Monterey, CA, USA. June 1980.

6. Anderson, E. A., Irvine, C. E., and Schell, R. R., “Subversion as a Threat in Information Warfare,” Journal of Information Warfare, Volume 3, No. 2, June 2004, pp. 52-65.

7. Lack, L., “Using the Bootstrap Concept to Build an Adaptable and Compact Subversion Artifice,” Master’s Thesis, Naval Postgraduate School, Monterey, CA, USA, June 2003.

8. Murray, J., “An Exfiltration Subversion Demonstration,” Master’s Thesis, Naval Postgraduate School, Monterey, CA, USA, June 2003.

9. Rogers, D., “A Framework for Dynamic Subversion,” Master’s Thesis, Naval Postgraduate School, Monterey, CA, USA, June 2003.

10. Anderson, E.A., *A Demonstration of the Subversion Threat: Facing a Critical Responsibility in the Defense of Cyberspace*, Master’s Thesis, Naval Postgraduate School, Monterey, CA, USA, March 2002.

11. Common Criteria Project Sponsoring Organizations (CCPSO). *Common Criteria for Information Technology Security Evaluation*. Version 3.0 Revision 2, CCIMB-2005-07-[001, 002, 003]. June 2005.

12. U.S. Department of Defense, “Trusted Computer System Evaluation Criteria,” DoD 5200.28-STD, 26 December 1985.

13. Anderson, J. P., Computer Security Technology Planning Study, ESD-TR-73-51, Vol. I, ESD/AFSC, Hanscom AFB, Bedford, Mass., October 1972 (NTIS AD-758 206).

14. Bell, D.E., and La Padula, L.J., "Secure Computer System: Unified Exposition and Multics Interpretation," MTR-2997, Mitre Corp., Bedford, MA, July 1975.

15. The Flux Research Group: Department of Computer Science, University of Utah, "The OSKit: The Flux Operating System Toolkit Version 0.97," March 2002. Available: http://www.cs.utah.edu/flux/oskit/html/oskit-www.html Accessed: June 2007.

16. Red Hat, Inc., "Red Hat Linux Reference Guide," 2003. Available: http://www.redhat.com/docs/manuals/linux/RHL-9-Manual/ref-guide/s1-grub-installing.html Accessed June 2007.

17. Subversion 1.4.4 – Available: http://www.redhat.com/docs/manuals/linux/RHL-9-Manual/ref-guide/s1-grub-installing.html Accessed June 2007.

18. APR – Available: http://svn.apache.org/repos/asf/apr/apr/branches/0.9.x Accessed June 2007.

19. APR-util – Available: http://svn.apache.org/repos/asf/apr/apr-util/branches/0.9.x Accessed June 2007.

20. OSKit 20020317 – Available: ftp://flux.cs.utah.edu/flux/oskit/oskit-20020317.tar.gz Accessed June 2007.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, VA

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, CA

3. Dr. Diana Gant
   National Science Foundation

4. Dr. Ralph Wachter
   ONR
   Arlington, VA

5. Dr. Cynthia E. Irvine
   Naval Postgraduate School
   Monterey, CA

6. Timothy Vidas
   Naval Postgraduate School
   Monterey, CA

7. Thuy D. Nguyen
   Naval Postgraduate School
   Monterey, CA

8. Donald Carter
   Civilian, Naval Postgraduate School
   Monterey, CA